CONCEPTUAL OBJECT-ORIENTED PROGRAMMING

201

by

TIMOTHY R. HINES

B.S., Kansas State University

A MASTER'S THESIS

Submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

Approved by:

Major Professor

Table of Contents

Chapter Page

List of Figures

Figure | Page

Ackowledgement

This paper was completed with the help of Dr. Elizabeth Unger whom I am unduly grateful and to Mary Lou for reading this thesis many many times.

1
Introduction

The growing demand for expert knowledge systems requires that we develop a language to fulfill the systems developers' needs. Most expert knowledge systems are restricted because of the languages chosen to implement them. Researchers in knowledge representation are looking for better ways to develop expert knowledge systems. The trend of programming languages has been leaning toward modular programming and the integration of declarative and procedural knowledge. The main characteristics of any programming language are its representation method or syntax, the semantics of its representation, how knowledge or data is stored and used, and its ability or intelligence to reason with its knowledge. In section 1.1 knowledge will be defined and several representational forms will be introduced. In section 1.2 intelligence will be discussed, and in chapter 2 the representation and semantics of the knowledge and language being used in this research will be presented. Chapter 3 will discuss modular programming and chapter 4 will define the formal syntax for a conceptual object-oriented programming language.

1.1 Knowledge and its Representational forms

What is knowledge? The simplest explanation of knowledge is that it has two types. The first type is

"Declarative Knowledge" -- real world facts that represent some type of truth. Philosophers have been working on the study of declarative knowledge (such as predicate logic) for many years. The philosophers Aristotle, Frege, Russell, Whitehead, and others have made important contributions to the understanding of knowledge.

Declarative knowledge can be defined to be true either in terms of observations, through proof procedures, or by hypotheses. A fact such as "Fred's grass is green" may have been observed by Mike, Fred's neighbor. We could represent this statement in the form of predicate logic. This would be represented as

green(grass) ^ own(Fred, grass)

Here we show that the grass is green and it is owned by Fred. If we want to know if Fred's grass is green, we could provide a program with the question, "Is Fred's grass green?." The sentence would be transformed into predicate logic and some sort of reasoning (such as resolution, unification, or other reasoning mechanisms) would be performed on the facts in the database. So far we have little information about the world surrounding Fred and Mike. Let us say that Fred and Mike both live in Kansas. When winter comes Fred's grass would probably turn brown, making the previous observation made by Mike false. We have shown from this that there are

facts that have short-term truth and facts that have long-term truth. The distinction between short-term and long-term truth is a problem when reasoning is performed on existing facts. We may need to add more knowledge about the surrounding world along with a set of procedures that could perform some sort of reasoning on them. For example, we might add the following rules

location(grass, Kansas) ^ time(winter) --> brown(grass)
location(grass, Kansas) ^ time(spring) --> green(grass)

Suppose we now wanted to find out if Fred's grass is green or brown. We could check to see where the location of Fred's grass is, and what time of year it is. Therefore, the above rules enable us to prove whether or not Fred's grass is green or brown.

A clearer proof procedure would be to prove 2 + 2 = 4. Through a set of algebraic rules we could prove this simple mathematical computation.

Another way to ascertain truth is to hypothesize. We, as humans, tend to do a great deal of hypothesizing, sometimes just on simple hunches. To hypothesize a statement let's say, for example, "President Ronald Reagan has false teeth." We think this statement is true because most people at his age have lost all their permanent teeth. It is still possible, and we would not know unless we asked him, that he may have some or all of his permanent teeth. Until we have ample proof we

can leave this statement as a hypothesized fact of truth.

The second type of knowledge is "Procedural Knowledge". We have stated previously that we can represent some knowledge in terms of facts. There are still other types of knowledge. These are acts that are performed as a set of procedures. We may know about something but we still need to know how to use it. For example, let's say we want to implement the blocks world for a robot. The robot's actions enable it to manipulate a set of blocks on a table. We could describe its manipulations as separate procedures, and the positions of the blocks as known facts. The robot can now change its blocks world by using what it knows, that is, its procedures and facts about the blocks.

```
                    +-------+
                    |   A   |
                    |       |
                    +-------+
                    +-------+
                    |   B   |
                    |       |
                    +-------+
                    |   C   |
                    |       |
                    +-------+
--------------------------------------------------
```

Figure 1.1.1 Initial state of the blocks world

We start with a set of facts that includes the following, block A is supported by block B, block B is supported by block C, and block C is supported by the table, shown in figure 1.1.1.

```
+-------+
|   B   |
|       |
+-------+
+-------+    +-------+
|   C   |    |   A   |
|       |    |       |
+-------+    +-------+
-----------------------------------------------------
```

Figure 1.1.2 Intermediate state of the blocks world

```
+-------+    +-------+    +-------+
|   A   |    |   B   |    |   C   |
|       |    |       |    |       |
+-------+    +-------+    +-------+
-----------------------------------------------
```

Figure 1.1.3 Final state of the blocks world

We want the robot to perform the simple task of placing block B onto the table, as shown in figure 1.1.3. To perform the task the robot would have to pick up block A, put it on the table, pick up block B, and then put it on the table. The necessary procedures for the robot would include reasoning with the known block positions, picking up a block, and putting a block down. Before the robot can perform any operations it must first determine what its world looks like. Using figure 1.1.1 the robot finds out that to get block B on the table, it must first remove block A from its position. When block A is taken from its position and placed onto the table, shown in figure 1.1.2, the robot looks at its world again to see what it must do next to get block B onto the table. The robot finds that it

only needs to pick up block B and put it onto the table, thus completing its requested task.

We have described two types of knowledge, declarative and procedural knowledge. In the past there have been arguments saying that we could represent all the procedural knowledge in a declarative representational form. The Procedural/Declarative controversy of the seventies addressed this issue. The following arguments of the discussion were presented in Winograd[1975]. The declarativists argued that each fact would be stored once, regardless of the number of different ways it may be used. New facts are easily added without changing other facts or small procedures. The proceduralists declared that it is easy to represent knowledge of how to perform tasks. It is also easy to represent knowledge that does not fit well in declarative representations, such as default and probabilistic reasoning. Procedures also represent heuristic knowledge that does efficient problem-solving. Exceptions that occur, these are problems not defined by formal theory, can be handled separately in procedures. The declarative representation may find it difficult to handle exceptions not included within its knowledge.

One agreement that came out of the Procedural/Declarative controversy was that we needed to find some sort of bridge between the two types of

knowledge representation. We need to integrate declarative and procedural knowledge. Much of the early work attempting to do this was through the use of semantic networks and frames. Research using semantic networks has pretty much been abandoned. But the use of frames has been more promising. Knowledge representation languages and tools that include frame representations, have recently been developed. Some of these are KEE, KRYPTON, and others.

It has been shown how predicate logic could represent some types of knowledge. Current work in knowledge representation has centered around frames and efficient reasoning methods. All the types of knowledge that have been described must be represented in some representational form which will retain most of its meaning. Also each form must be able to represent both declarative and procedural knowledge. The representational forms presented here can be placed within the knowledge levels. The knowledge levels consist of five types: language, conceptual, epistemological, logical, and physical.

At the physical level, the lowest level in the knowledge levels, we could represent knowledge in the form of bits in a computer (zeros and ones). For example, 0001000100101 may represent a symbol in the English language. This representational form is not

very meaningful to us as humans, it is very hard to read, and only through an interpreter could we make any sense out of it.

The next level up is the logical level. Most programming languages reside at this level. Languages such as Lisp, Pascal, and PL/I are logical representational languages.

The epistemological level begins to represent the knowledge in a more meaningful way. Prolog can be placed somewhere near this level, although it still resembles that of a logical language. Prolog is a rule-based language that uses a predicate logic theorem prover.

At the conceptual level representational forms begin to retain more meaning of the knowledge. Some of these representational forms are semantic networks, scripts, frames, and conceptual dependency.

Semantic networks are the earliest form of structured representation of knowledge. They were first developed by Quillan and Raphael in 1968. Semantic networks have been designed to be able to describe both events and objects. The information in semantic networks is represented as a set of nodes which are connected together by a set of arcs. These arcs represent relationships between the nodes. An example of a seman-

tic network would be the following, shown in figure 1.1.4, using the sentence "John gave the book to Mary."

```
                          +-----+
                          |Give |
                          +-----+
                             ^
                             |
                             |ISA
                             |
+-----+    AGENT          +-----+   OBJECT       +-----+
|John |<----------| EV7  |----------->|BK23 |
+-----+                   +-----+                +-----+
                             |                      |
                             |BENEFICIARY           |ISA
                             |                      |
                             v                      v
                          +-----+                +-----+
                          |Mary |                |Book |
                          +-----+                +-----+
```

Figure 1.1.4 Semantic network for "John gave the book to Mary."

It is possible for us to represent all the knowledge in the semantic network as two-place predicates in predicate logic. For example, figure 1.1.4 would have the following two-place predicates.

```
ISA(Book,BK23)
AGENT(EV7,John)
OBJECT(EV7,BK23)
ISA(EV7,Give)
BENEFICIARY(EV7,Mary)
```

Semantic networks have been deeply explored but have given way to better and more powerful structured knowledge representation forms.

Scripts are another form of representing structured knowledge. Scripts are used to describe a sequence of events. A script contains a set of slots and each slot may contain some information about the values that the slot may contain. Each of the slots may also contain a default value that will be used when a value for a slot is not available. The components of a script consist of entry conditions, results, props, roles, track, and scenes. Scripts seem to be useful because they can easily represent a recurring sequence of events. The sequence of events in a script form a causal chain. The beginning of the chain is the event's entry conditions and the end the chain is the results of the script. Events contained in the causal chain are connected to earlier events that make these events possible to occur and to later events that let them occur. Figure 1.1.5 and figure 1.1.6 show a sample script, that of a library.

```
+--------------------------------+
|Script: Library                 |
|Track: College                  |
|       Library                  |
|Props: C = Card Catalog         |
|       Books                    |
|       CRT                      |
|       ID = Student ID          |
|       CCN = card catalog number|
|       AI = author index        |
|       SI = subject index       |
|       TI = title index         |
|       SU = subject             |
|                                |
|Roles: S = student              |
|       L = Librarian            |
|                                |
|Entry Conditions:               |
|       S drops off book(s)      |
|        and/or                  |
|       S is checking out books  |
|       S has ID                 |
|                                |
|Results: S has no books         |
|          or                    |
|         S has new book(s)      |
+--------------------------------+
```

Figure 1.1.5 Script for a library scene.

```
+-----------------------------------------------+
|Scene 1: entering                              |
|               S enters library.               |
|                  |        |                  |
|                  |        |                  |
| S drops off book(s).  S moves to C.          |
| (go to scene 2) or    (go to scene 2)        |
| (go to scene 5)                               |
|-----------------------------------------------|
|Scene 2: Choose book(s)                        |
| (knows author)  (knows subject)  (knows title)|
| S moves to AI.  S moves to SI.   S moves to TI.|
|      |          S thinks of SU.       |       |
| S pulls out C.  S pulls out C.   S pulls C.   |
|      |     |        |        |       |   |    |
|      |     |        |        |       |   |    |
|      |     +-----> | <-----|---> | <-+       |
|      |             |       v       |          |
|      +-------------+---------------+          |
|                    |       |                  |
|                    |       |                  |
|                    v       v                  |
|  S does not find book.   S writes down CCN.   |
|  (go to scene 2) or      S goes to stacks.    |
|  (go to scene 4) or                           |
|  (go to scene 5)                              |
|-----------------------------------------------|
|Scene 3: Stacks                                |
|  S chooses correct stack.                     |
|     |                   |                     |
|     |                   |                     |
| S finds book.      S does not find book.     |
|              |          |                     |
|          (go to scene 2) or                   |
|          (go to scene 3) or                   |
|          (go to scene 4)                      |
|-----------------------------------------------|
|Scene 4: Check out                             |
|   S gives book(s) and ID to L.                |
|   L enters ID and CCN onto T.                 |
|   L gives ID and book(s) to S.                |
|-----------------------------------------------|
|Scene 5: Exiting                               |
|   S goes out of the library                   |
+-----------------------------------------------+
```

Figure 1.1.5 Script for a library scene.

Scripts seem to be quite effective for representing specific kinds of knowledge that is represented as a sequence of events. But scripts are less general than frames, therefore making scripts unsuitable for representing all the different kinds of knowledge.

Frames have recently been researched extensively and have been added to many knowledge representation systems. Frames, introduced by Marvin Minsky in 1975, describe classes of objects. A frame contains a collection of semantic net nodes and the slots together describe an object, act, or event. Each slot may have conditions that must be met before the slot can be filled. The slots may also contain a default value to be used when no information is available. Slots can contain procedural information called procedural attachments. Some types of procedural attachments are IF-ADDED, IF-REMOVED, and IF-NEEDED. Frames can be used to represent other types of knowledge, such as perspectives, prototypes, and individuals. Figure 1.1.6 shows an example of a how an object could be represented in the frame representation form.

```
(HURRICANE-BETTY
    (PLACE       (MIAMI-FLORIDA))
    (DAY         (5-23-35))
    (INJURED     (256))
    (FATALITIES  (6))
    (DAMAGE      (4235000))
```

Figure 1.1.6 A Frame that is an instance of type Hurricane with some of its slots filled.

Thus frames are like semantic networks. They are general-purpose structures in which particular sets of domain-specific knowledge can be embedded.

Conceptual dependency is the theory of representing the meaning of natural language sentences. From the representation we should be able to draw inferences from the sentences and its representation should be independent of its natural language. The conceptual dependency representation of a sentence is formed using conceptual primitives to form the meaning in the natural language sentence. One theory was first described by Roger Schank in 1973. His theory said conceptual dependency provides a structure and a set of primitives from which information can be constructed. Schank's primitives, listed in figure 1.1.7, show the primitive actions which may be used in a way to form high-level meaning in its representation. An example of Schank's theory is shown in figure 1.1.8.

ATRANS---Transfer of an abstract relationship
PTRANS---Transfer of the physical location of an object
PROPEL---Application of physical force to an object
MOVE-----Movement of a body part by its owner
GRASP----Grasping of an object by an actor
INGEST---Ingesting of an object by an animal
EXPEL----Expulsion of something from the body of an animal
MTRANS---Transfer of mental information
MBUILD---Building new information out of old
SPEAK----Producing sounds
ATTEND---Focusing of a sense organ toward a stimulus

Figure 1.1.7 Schank's primitive actions

```
                                        to
                                     +----> Mary
          p               o        R |
    John <===> ATRANS <--- book <---|
                                     |from
                                     +----< John
```

Figure 1.1.8 Conceptual dependency using Schank's primitives. It represents the sentence "John gave Mary the book."

The arrows of the conceptual dependency in figure 1.1.8 indicate the direction of dependency; double arrows indicate a two way link between actor and action; p indicates past tense; ATRANS indicates transfer of possession and is also one of the primitive actions; o indicates the object case relation; and R indicates the recipient case relation.

Schanks' representational form seems to represent its knowledge at a low-level. The feeling of representing knowledge at a low-level is apparent because of the low-level primitive actions used when representing natural language sentences. Also the method does not

seem to be very easy to use because of its notation and the use of the types attached to the conceptual dependency.

Another conceptual dependency representation method is John Sowa's theory of conceptual graphs. There is no set of primitive actions like Schank's method. Natural language sentences are converted to the conceptual graphs in a way that allow the graphs to retain the high-level meaning of the sentence. Conceptual graphs have been chosen as the knowledge representation method used to represent the knowledge in this research. Conceptual graphs will be discussed in detail in chapter 2.

The last representational form in the knowledge levels is language. We could choose language as our representational form. However, there are no natural language parsers which have been fully implemented that will allow us to convert from, say English, to a programming language or a representational form that could be used by the computer. Therefore, the next best representation level is the conceptual level. Although at the conceptual level the knowledge represented by the user is an arbitrary decision, it is possible to set standards when creating new knowledge in the conceptual notation.

It is the intent of this research to use a uniform knowledge representation method for both procedural and declarative knowledge, integrating the two types of knowledge together. In this research the same representation method as used by Hartley[1985] has been chosen. John Sowa's conceptual graph theory is used to represent both declarative and procedural knowledge. The conceptual graph theory will be discussed in greater detail in chapter 2, showing the formal notation and how the graphs are applied to real world knowledge.

1.2 Intelligence

Am I intelligent? Are you? Can a machine be intelligent? These questions and more are researched in the fields of artificial intelligence and cognitive science. Usually, when we try to define intelligence we start out by trying to understand the human mind, how it works, and what makes it work. This has proved to be a long and enduring project for many researchers in all scientific fields. Intelligence can be described in many ways including how well an entity can adapt to new environments. The entity has the ability to reason with its existing knowledge, to come up with an optimum solution to fulfill its present needs. This ability to reason can be present in varying degrees. For example, children at many times have difficulty in determining a

correct answer to a problem or the correct behavior in a particular situation. Many of these problems stem from the childs' lack of knowledge, but even if the knowledge is present the child still may not have the ability to reason or to even understand the problem.

If intelligence can be construed as the capacity to learn, what gives us that capacity, and how much of a capacity is needed to solve complex problems? To begin with, an entity must contain a certain amount of common-sense. This helps the entity to reason with its existing knowledge. The amount of common-sense will help with the extraction of useless answers to the questions. Therefore, the entity can contain varying degrees of common-sense.

Another factor that is important to intelligence is the ability to understand the knowledge that is contained within the entity. For example, many college students pass classes by memorizing written knowledge in a text book. Students who only memorize the material in their text books may never understand what they have read. On the other hand, many college students take the time to remember the knowledge plus they try to understand what that knowledge means. The college students that memorize text books will have a difficult time if they are asked questions relating to the subject that they have memorized. It is believed that the

students who have understood the knowledge will have a better chance of answering the same set of questions correctly.

Intelligence can now be defined in terms of our ability to reason, the amount of common-sense available, and the understanding of stored knowledge. These are factors that AI researchers hope to include within their own AI expert knowledge systems. Within this research, it is the intent to try to include some of the intelligent characteristics present in the human mind. In the past there have been systems built that replicate common-sense, but the ability to reason and to understand the knowledge are still problems, as pointed out by Schank and Childers[1984].

Intelligence is a composite or combination of human traits, which includes a capacity for insight into complex relationships, all of the processes involved in abstract thinking, "adaptability in problem solving, and capacity to acquire new capacity", Cattell[1971].

1.3 The Problem

Researchers in artificial intelligence have been searching for ways to design languages and systems that allow the modeling of the knowledge and intelligence necessary for expert problem solving. When implementing

an expert knowledge system there is a need for ways to reason with the knowledge, and a high-level representational form that is uniform over both procedural and declarative knowledge.

Most languages and tools do not provide the ability to clearly program in a variety of reasoning methods thus allowing the flexibility necessary for implementation of expert knowledge systems. There are but a few systems that incorporate multiple reasoning methods. Some of these are KEE, MRS, ART, and LOOPS. The reasoning methods allowed are antecedent-consequent, logical, heuristic, and plausible reasoning. Research is still advancing in this area, but no one has yet provided a uniform syntax.

Most conventional procedural programming languages such as Pascal, Fortran, Pl/I, do not contain a uniform notation that will elevate procedural knowledge to the same representational level as declarative knowledge, Hartley[1985]. A programming language such as Pascal has a set of commands, and these commands can be used to store and manipulate knowledge. The commands and the knowledge can be represented in many different ways making it necessary to learn and know all of the notations. The great demand for developing systems in a short amount of time, or at least for developing a prototype, has made it clear that these procedural level

languages will not fulfill the necessary requirements.

Knowledge representation researchers are working on the development of high-level knowledge representation languages. The problem arising out this research is the actual representation of both the declarative and procedural knowledge. Even though frames seem to be quite adequate in many ways, they still do not represent all types of knowledge necessary in expert knowledge systems. For instance, representing a situation that is possibly true can be quite difficult. The representation of an animate object experiencing a state may also be difficult when representing it in frames. Therefore, the search goes on for the perfect representational form, or forms, needed to represent knowledge without losing any meaning.

There have been other programming languages and tools developed to aid in the implementation of expert knowledge systems. These include Flavors, Loops, and CommonLoops which have been designed to provide the flavor of object-oriented programming. These languages have yet to be shown useful in completely implementing all of the knowledge necessary for development of expert knowledge systems. However, the most promising of these languages will probably be CommonLoops because of the widespread concern and support to standardize object-oriented programming languages.

Researchers have concluded that there is a need to integrate procedural and declarative knowledge with the addition of modular programming constructs and techniques to use the knowledge. This is the problem addressed by the model and method proposed in this paper.

1.4 Solution Approach

The approach is a high-level representational method, uniform over both procedural and declarative knowledge, with object-oriented programming constructe. The knowledge representation method chosen to represent the declarative and procedural knowledge is John Sowa's conceptual graphs. The representational form of conceptual graphs elevates the declarative knowledge to the same level as procedural knowledge. The object-oriented style of programming is a combination of Loops, CommonLoops, Flavors, and KEE. There are pros and cons associated with each system. We attempt to extract the best features from each of these object-oriented languages and combine them into a uniform system.

Conceptual Graphs

Perceptions made by humans form mental models in response to some external entity or scene. The mental models consist of percepts, the matching of each percept to some part of the input, and a conceptual graph that shows how the percepts relate to one another. For every percept there is a concept that is the interpretation of that percept. The percept is the image of a concept. Some concepts do not have any images. The concept nodes in the conceptual graphs represent entities, attributes, states, and events, and the relation nodes show how the concept nodes are related to each other. Chapter 2 describes John Sowa's conceptual graph theory. The notation, the six types of conceptual graphs, and how they may be used in a database system will be discussed within this chapter. During the discussion of the conceptual graph theory, the reader will discover its ability to represent knowledge at the conceptual level as discussed in section 1.1 on knowledge and its representation.

2.1 Notation

Conceptual graphs can be represented in two ways, either by diagrams which are drawn in a display form as linked boxes and circles, or in a linear form using boxes represented by square brackets "[]", and circles

represented by rounded parentheses "()". The boxes or brackets represent concepts and the circles or rounded parentheses represent conceptual relations. The linear conceptual graph notation will be used for all the examples presented in this paper. The following example is a conceptual graph of the sentence "A fiddler sitting on a roof."

```
[FIDDLER] <- (AGNT) <- [SIT] -> (LOC) -> [ROOF].
```

The concepts in above example are [FIDDLER], [SIT], and [ROOF]. The relation nodes are agent (AGNT) and location (LOC). The period "." at the end of the graph terminates the entire conceptual graph. The next example shows how other special symbols are used in the conceptual graphs. The sentence "A monkey eating a walnut with a spoon made out of the walnut's shell" would be the following conceptual graph.

```
[EAT]-
   (AGNT) -> [MONKEY]
   (OBJ)  -> [WALNUT: *x]
   (INST) -> [SPOON] -> (MATR) -> [SHELL]-
                     (PART) <- [WALNUT: *x].
```

The symbol *x is a variable. It represents an unspecified individual of the concept type WALNUT. In the sentence there are no individuals known at this time, but in the conceptual graph the two concept nodes [WALNUT]

must have the same individual referent. The reason for the variable is because the graph contains a cycle and any graph that contains cycles needs a variable to show cross reference. The graphs says that there is an agent (AGNT) that is performing the act [EAT]. The object (OBJ) of the act [EAT] is a [WALNUT] and the instrument (INST) being used for the act [EAT] is a [SPOON]. The [SPOON] has a material (MATR) that is made from a [SHELL] and the [SHELL] is part (PART) of a [WALNUT] which is the same [WALNUT] that the [MONKEY] is eating. The hyphen "-" shows that the relations that are connected to the concept type [EAT] are listed on subsequent lines. The hyphen and the comma ",", shown in the next example, form a bracketing pair that is necessary for linearizing the graph.

When choosing a concept as the head it is best to pick the concept that has the most relations connected to it. If we had chosen the concept [SPOON] as the head, the graph would be represented as:

```
[SPOON]-
  (INST) <- [EAT]-
             (OBJ)- -> [WALNUT] -> (PART) -> [SHELL: *y]
             (AGNT) -> [MONKEY],
  (MATR) -> [SHELL: *y].
```

The comma in the example ends the connection of relations to the concept [EAT]. This is so the relation

(MATR) is linked to the concept [SPOON] and not to the concept [EAT]. The *y is a variable used as the same individual for both concept types [SHELL].

In the graphs it is possible to represent a specific individual of a concept type rather than an unspecified individual. To represent a specific individual of the concept [STUDENT] we would write it as [STUDENT: #512643635]. The "#" symbol followed by an integer represents the definite referent of the concept STUDENT. If the student's name is known it can be added to the concept STUDENT. A relation node (NAME) would be connected to the concept [STUDENT] and a concept node containing the string of the student's name would be linked to the relation node (NAME). This graph would be written as the following:

[STUDENT: #512643625] -> (NAME) -> ["Tim Hines"].

The above conceptual graph can be abbreviated. The relation (NAME) can be eliminated and the string "Tim Hines" would be placed between the ":" and the "#" symbols. The abbreviated graph is shown below.

[STUDENT: Tim Hines#512643625].

Given the sentence "A gold bar weighs 400 ounces" a conceptual graph could be constructed. The concepts of the sentence are [GOLD], [BAR], and [WEIGHT]. The

concept [WEIGHT] would have another concept attached to it. The concept [MEASURE] shows the measure of the gold bar. The concept [MEASURE] has a relation (NAME) connected to it and (NAME) has a concept connected to it that represents the measure's name. The graph would be written as the following:

```
[GOLD] <- (COLR) <- [BAR] -> (CHRC) -> [WEIGHT]-
      (MEAS) -> [MEASURE] -> (NAME) -> ["400 ounces"].
```

In the student individual example we removed the relation (NAME). The above example can also be abbreviated in the same manner. The abbreviated graph would be the following:

```
[GOLD] <- (COLR) <- [BAR] -> (CHRC) -> [WEIGHT]-
      (MEAS) -> [MEASURE: 400 ounces].
```

Since the concept node [MEASURE] is a common concept and used quite often, the above graph can be abbreviated. The concept [MEASURE] can be eliminated and the string "400 ounces" representing a specific measure can be placed in the concept [WEIGHT]. This is done by using the symbol "@", where the string immediately following the "@" is a numerio number with a symbol representing the meaning of that number. In the example

below the number 400 is placed after the "@" symbol with the string "ounces" as the symbol representing the meaning of the number 400. The following example shows the measure placed in the concept [WEIGHT].

```
[GOLD] <- (COLR) <- [BAR]-
      (CHRC) -> [WEIGHT: @400 ounces].
```

Conceptual graphs can have generic sets containing zero or more elements referent to the concept type. For the sentence "Beth sees some students" the graph would be written as shown in the following example. The symbol {*} in the concept node [STUDENT] represents a generic set of students where the individuals are not known.

```
[PERSON: BETH] <- (AGNT) <- [SEES]-
      (OBJ) -> [STUDENT: {*}].
```

A specific set of students could be referenced by the graphs. The definite set of individuals would be a conjunctive set where each individual in the set is separated by a comma. In the sentence "Beth sees the students Randy and Tim" the graph would be represented below.

```
[PERSON: Beth] <- (AGNT) <- [SEES]-
      (OBJ) -> [STUDENT: {Randy,Tim}].
```

A graph may contain a set that contains definite individuals and a generic set. The sentence "Beth sees four students, Randy, Tim, and two others", would be represented as in the example below. The @4 says that there are 4 students of which only two are known.

```
[PERSON: Beth] <- (AGNT) <- [SEES]-
     (OBJ) -> [STUDENT: {Randy,Tim,*}@4].
```

A disjunctive set can also be represented in the conceptual graphs. For the sentence "The martian Zoolu lives on either Mars or Venus" the disjunctive set notation is represented in the following graph. The elements within the brackets "{}" would be separated by a vertical bar instead of by commas. The disjunctive set is the set Mars or Venus.

```
[MARTIAN: Zoolu] -> (STAT) -> [LIVE]-
     (LOC) -> [PLANET: {Mars|Venus}].
```

There are two other types of sets that can be formed in the conceptual graphs. They are the distributive and respective sets. A distributive set in a conceptual graph can be shown by the sentence "Two students each read three books". The keyword DIST is used immediately proceeding the set in the concept type.

```
[STUDENT: DIST{*}@2] <- (AGNT) <- [READ]-
      (OBJ) -> [BOOK: {*}@3].
```

The respective set uses the keyword RESP immediately proceeding the sequence delineated by angle brackets. The sentence "John, Dick, and Harry read three books, Old Yeller, Foundation, and The Prophet, respectively" would be represented by the following graph.

```
[STUDENT: RESP<John,Dick,Harry>] <- (AGNT) <- [READ]-
  (OBJ) -> [BOOK: {Old Yeller,Foundation,The Prophet}].
```

A summary of the notation used in the conceptual graphs is listed below.

* Concept nodes are represented with square brackets as in the concept [PERSON].
* Relation nodes are represented with rounded parentheses. The relation (AGNT) would be the agent of some concept.
* Concepts are connected to relations with arrows to form conceptual graphs. The following graph says that the relationship of CONCEPT1 is CONCEPT2. [CONCEPT1] -> (REL) -> [CONCEPT2]
* The symbol *x is a variable representing an unspecified individual of a concept node. The variable name can be any combination of letters such as *abc or *name.
* The hyphen "-" allows for relations to be listed on subsequent lines.
* The comma "," terminates a hyphen.
* The period "." terminates the entire graph.
* The "#" symbol followed by an integer represents a definite referent of a concept node.

* The "@" symbol is followed by a number and a string that represents the meaning of the number. This number and string represents a specific measure for a concept. The string is optional. If it is not present then the number after the "@" is taken as the number of items of that type.

* The symbol {*} represents a generic set of zero or more elements referent to a concept node.

* The conjunctive set {John,Mike,Harry} represents a specified set of individuals referent to a concept node, where each element in the set is separated by a comma.

* The disjunctive set {Mars|Venus} represents the set where either Mars or Venus is true. Each element in the set is separated by the vertical bar "|".

* The symbol "DIST" represents a distributive set as in DIST{John,Harry}.

* The symbol "RESP" represents a respective set. The elements of the set are enclosed within angle brackets such as RESP<John,Harry>.

The notation for the conceptual graphs has been provided in this section. In the following sections canonical graphs, graphs that represent possible situations, the six types of conceptual graphs that can be formed, and the rules that may be applied to the graphs are discussed.

2.2 Canonical Graphs

A conceptual graph contains concept nodes and relation nodes where every concept in the graph is linked to a relation node. There is an infinite number of combinations of conceptual graphs, but not all of

them represent a real or possible situation. The sentence "Colorless green ideas sleep furiously" could be represented by the following graph.

[SLEEP] -> (AGNT) -> [IDEA] -> (COLR) -> [GREEN].

The graph is a valid conceptual graph, but it does not reresent a real-world situation. To eliminate the absurd combinations of conceptual graphs, selectional constraints are applied to valid conceptual graphs. The constraints that are applied are observation, derivation, and insight. In the real-world we perceive events and states. The observations that are perceived are recognized as canonical graphs. In derivation, new canonical graphs may be formed using formation rules that are applied to existing canonical graphs. Insight or creativity may be used to form new canonical graphs that extend or replace old canonical graphs. This would be done when a canonical graph did not adequately fit a specific situation. The three constraint rules, observation, derivation, and insight are the same as the three types of knowledge acquisition that were discussed in section 1.1., these knowledge acquisition types were observation, proof procedures, and hypotheses.

2.3 Six forms of Conceptual Graphs

There are six forms that can be produced using

conceptual graph theory. These are: Type -- which is like function definition in Lisp, it is a method of assigning a label to an abstraction; Relation -- defining the relationship between entities; Individual -- a specific instance of a concept type; Prototype -- a generic individual of a concept type; Schema -- the basic structure for representing background knowledge for human-like inference; and Actor -- a process that responds to messages by performing some service and then generating messages to pass onto other actors or concepts. In the following sections, the six forms of conceptual graphs will be discussed and examples given for each.

2.3.1 Type

Type definitions are based on Aristotle's definition of genus and differentia which allows types to be defined and placed within a type hierarchy. In the type definition, some concept is chosen as the genus and a canonical graph is the differentia. The syntax for a type definition would be written as:

```
type type-name(argument) is
        canonical-graph.
```

The type-name is an abstraction of some concept, type-name(argument) is called the genus of the type, and the canonical graph is called the differentia of the type-

name. In the type definition any generic concept in the canonical graph may be chosen as the genus for the type definition.

The following type definitions are based on the same differentia but have different concepts as the genus. The three graphs show how the concept ELEPHANT can have different type labels. The first defines the type label CIRCUS-ELEPHANT, an ELEPHANT that performs in a CIRCUS, as a subtype of ELEPHANT.

```
type CIRCUS-ELEPHANT(x) is
     [ELEPHANT: *x]-
        (AGNT) <- [PERFORM] -> (LOC) -> [CIRCUS].
```

The canonical graph says there is an ELEPHANT that is the agent (AGNT) of the action PERFORM. The ELEPHANT has a location (LOC) which is in the CIRCUS. The subtype of ELEPHANT is CIRCUS-ELEPHANT where x is its argument. The variable *x is the referent to the concept ELEPHANT. In section 2.3.2 we will discuss individuals where the argument of a type-name will be filled by a specific value.

The other two concept nodes in the canonical graph could have been chosen as the genus for the type label rather than ELEPHANT. Choosing another concept as the genus would produce a different type definition. The following graph represents a type definition where the concept CIRCUS has been chosen as the genus. The graph

says that there is a circus which has elephants performing some act in that same circus.

```
type ELEPHANT-CIRCUS(y) is
   [ELEPHANT]-
      (AGNT) <- [PERFORM] -> (LOC) -> [CIRCUS: *y].
```

Had PERFORM been chosen as the type label the following graph would be formed. The graph says that there are performances [PERFORM] that ELEPHANT's do, and the performances [PERFORM] have a location (LOC) which is in a [CIRCUS].

```
type ELEPHANT-PERFORMANCE(z) is
   [ELEPHANT]-
      (AGNT) <- [PERFORM: *z] -> (LOC) -> [CIRCUS].
```

As you can see from the previous type definitions, a type hierarchy becomes apparent. The type definitions CIRCUS-ELEPHANT, ELEPHANT-CIRCUS, and ELEPHANT-PERFORMANCE are subtypes of the concepts ELEPHANT, CIRCUS, and PERFORM, respectively. The diagrams below show how the type hierarchy may be formed.

```
   MAMMAL            PLACE            ACT
     |                 |               |
     |                 |               |
     v                 v               v
  ELEPHANT          CIRCUS          PERFORM
     |                 |               |
     |                 |               |
     v                 v               v
   CIRCUS           ELEPHANT        ELEPHANT
  ELEPHANT           CIRCUS       PERFORMANCE
```

2.3.2 Individual

An individual would be defined by filling in one or more of the generic concepts within the body of a type definition, where the generic concepts become individual concepts. The syntax for an individual is shown below. The type-name would be the same as in a type definition and the argument of the type-name is its value.

```
individual type-name(argument-value) is
        canonical-graph.
```

For an individual the type-name would need to be previously defined in a type definition. If it hasn't been previously defined, then an individual cannot be created. For the following example the type definition CIRCUS-ELEPHANT will be used.

```
type CIRCUS-ELEPHANT(x) is
   [ELEPHANT: *x] <- (AGNT) <- [PERFORM]-
                                (LOC) -> [CIRCUS].
```

If we used the CIRCUS-ELEPHANT type definition we could create an individual by filling in one or more of its generio concepts. To create the individual CIRCUS-ELEPHANT we must choose a specific elephant. In the individual definition, the elephant Jumbo has been chosen and the location is the Barnum & Bailey circus. The symbol {*} denotes a generic set of performances

that is performed by the elephant Jumbo.

```
individual CIRCUS-ELEPHANT(JUMBO) is
      [ELEPHANT: Jumbo] <- (AGNT) <- [PERFORM: {*}]-
           (LOC) -> [CIRCUS: Barnum & Bailey].
```

To use database concepts in conceptual graphs, a specific instance of a record type could be formed. For example we could describe the events that take place when reserving a room at a hotel. The type definition for the type name HOTEL-RESERVATION could be represented in the following example.

```
type HOTEL-RESERVATION(reservation-no) is
      [RESERVATION: *reservation-no]-
         (RCPT) -> [PERSON]
         (OBJ)  -> [ROOM] -> (LOC) -> [HOTEL]
         (DUR)  -> [TIME-PERIOD]-
                    (STRT) -> [ARRIVAL-DATE]
                    (UNTL) -> [DEPARTURE-DATE].
```

A specific individual of the type definition HOTEL-RESERVATION would be created by filling in the generic concepts within the body of the conceptual graph. A specific hotel reservation with a unique reservation number would be represented by the following individual conceptual graph.

```
individual HOTEL-RESERVATION(#4128) is
  [RESERVATION: #4128]-
   (RCPT) -> [PERSON: Tim Hines]
   (OBJ)  -> [ROOM: 234]-
               (LOC) -> [HOTEL: Doral Tuscany]
   (DUR)  -> [TIME-PERIOD: @ 2 night]-
             (STRT) -> [ARRIVAL-DATE: June 30, 1984]
             (UNTL) -> [DEPARTURE-DATE: July 1, 1984].
```

2.3.3 Relation

In the previous sections of chapter 2 we described different methods of forming conceptual graphs with concept and relation nodes. In this section original conceptual relations are discussed. These may be defined and used to form new conceptual graphs. The syntax for the relation definition would be written as the following.

```
relation relation-name(argument1,..,argumentn) is
                canonical-graph.
```

The relation-name is a unique name that is an abstraction over argument1 to argumentn, where n is the number of arcs attached to the relation-name when used in a new conceptual graph. The body of a relation definition must be a canonical graph that relates to the relation-name. The example below is a new relation called PAST which has only one argument. This means there is only one arc attached to the relation PAST when used in a new conceptual graph. The conceptual graph for the relation definition PAST says there is a situation that has occurred at some point in time where the successor of that time is now.

```
relation PAST(x) is
  [SITUATION: *x]-
      (PTIM) -> [TIME] -> (SUCC) -> [TIME: NOW].
```

For a database system we could describe the functions that occur within the database system. For example, the sentence "A part number is a characteristic of a set of items, and a number is the quantity of such items located in a stock room" could be represented by the following relation definition.

```
relation QOH(x,y) is
   [PART-NO: *x] <- (CHRC) <- [ITEM: {*}]-
                                 (QTY) -> [NUMBER: *y]
                                 (LOC) -> [STOCKROOM].
```

The relation definition of QOH has two formal parameters x and y. When the relation QOH is used in a conceptual graph it needs two concepts linked to it. The conceptual graph of the relation says there is a generic set of items denoted by the {*} symbol. The ITEM concept has a characteristic that is a part number, a location in the stockroom, and a quantity for the item. When a relation is used it may be contracted. The relation QOH may now be contracted since it has been previously defined. The graph given below is the contracted QOH relation.

```
[PART-NO] -> (QOH) -> [NUMBER].
```

All relation definitions used in the conceptual graphs may be reduced to a single dyadic relation type called LINK. For example, the relation (AGNT) that has been used quite extensively so far can be defined by the following relation definition. It is defined in terms

of a concept type AGENT.

```
relation AGNT(x,y) is
   [ACT: *x] <- (LINK) <- [AGENT]-
                          (LINK) -> [ANIMATE: *y].
```

2.3.4 Schema

Schemata are used to represent plausible combinations of conceptual graphs which enable human-like inference. A schema is a perspective of a situation, entity, or state. In turn these perspectives may form new perspectives. For example, the concept MAN may have several meanings depending on how the concept is used, such as HUMAN and HOMOSAPIEN. The collection of all these perspectives grouped together forms a schematic cluster. In section 1.1, we discussed two knowledge representation methods, frames and scripts, which are very similar to schemata. The syntax for a schema would be written as the following.

```
schema for concept-type(argument) is
       canonical-graph.
```

The next example is a schema for the concept type BUS. In the conceptual graph conditions may be used to delineate concepts. The condition for the concept SPEED states that the speed must be less than or equal to 55 miles per hour. There is also an approximation for the number of passengers contained within the bus. The

approximation is about 50 people. In the example below, the schema for BUS is only one perspective. There could be many more defined for it.

```
schema for BUS(x) is
  [BUS: *x]-
    (INST) <- [TRAVEL]-
                (RATE) -> [SPEED<=55mph]
                (AGNT) -> [PASSENGER: {*}]-
                             (QTY) -> [NUMBER~=50: *y],
    (CONT) -> [PASSENGER: {*}]-
                (QTY) -> [NUMBER~=50: *y]
    (OBJ)  -> [DRIVE] -> (AGNT) -> [DRIVER].
```

2.3.5 Prototype

A prototype is a generic or average individual. A prototype is derived by taking one or more schemata in a schematic cluster and generalizing to form an average schema, a "prototype". The prototype definition can contain default values as referents to their concepts. The syntax for a prototype is shown below.

```
prototype for concept-type(argument) is
            canonical-graph.
```

An example of a typical elephant could be represented by the following prototype definition.

```
prototype for ELEPHANT(x) is
   [ELEPHANT: *x]-
      (CHRC) -> [HEIGHT: @ 3.3 m]
      (CHRC) -> [WEIGHT: @ 5400 kg]
      (COLR) -> [DARK-GRAY]
      (PART) -> [NOSE]-
                  (ATTR) -> [PREHENSILE]
                  (IDNT) -> [TRUNK],
      (PART) -> [EAR: {*}]
                  (QTY) -> [NUMBER: 2]
                  (ATTR) -> [FLOPPY],
      (PART) -> [TUSK: {*}]-
                  (QTY) -> [NUMBER: 2]
                  (MATR) -> [IVORY],
      (PART) -> [LEG: {*}] -> (QTY) -> [NUMBER: 4]
      (STAT) -> [LIVE]-
                  (LOC) -> [CONTINENT: {Africa|Asia}]
                  (DUR) -> [TIME: @ 50 years].
```

In the example above, the prototype for elephant contains default values which are standards for an average elephant. The concept HEIGHT has a default value which is 3.3 meters which is the approximate height of an average elephant. With the prototype of an elephant, the definition is true of a typical elephant, but it may not be true with a specific elephant. For example, a baby elephant would not fit the prototype given above for a typical elephant.

2.3.6 Actors

The last type for defining conceptual graphs is an actor graph. An actor is initiated by a set of external messages which are called its inputs. The actor then performs its task and generates another set of messages called the outputs. The outputs are then passed onto other actors or concepts. The actor message is very

similar to message passing in object-oriented systems, which is discussed in chapter 3. When combining several actors together networks of dataflow graphs are formed. The actor graphs are then used for doing computation, solving complex problems, and/or simulating events and processes. The syntax for the actor graph is given below.

```
actor actor-name(input-args,output-args) is
        conceptual-graph.
```

An actor graph could be as simple as an arithmetic function like divide. An actor graph definition for divide is presented below. The actors use a different notation. They are represented by angle brackets "<>" which distinguishes them from concept nodes. The actor graph DIVIDE has two inputs, dividend and divisor, and two outputs, quotient and remainder. The arrows in the definition show which direction the data is flowing. The arrow pointing left from the concept [DIVIDEND: *a] shows that it is an input to the actor divide.

```
actor DIVIDE(in a,b; out c,d) is
      <DIVIDE>-
         <- [DIVIDEND: *a]
         <- [DIVISOR: *b]
         -> [QUOTIENT: *c]
         -> [REMAINDER: *d].
```

A more complex example of an actor graph is provided in the next actor graph definition which defines the recursive factorial function.

```
actor FACTORIAL(in n; out x) is
  [NUMBER: *n]-
    -> <IDENT> -> [NUMBER=0] -> <ADD1> -> [NUMBER: *x]
    -> <IDENT> -> [NUMBER>0]-
                    -> <MULTIPLY>-
                          <- [NUMBER: *z]
                          -> [NUMBER: *x],
                    -> <SUB1>-
                         -> [NUMBER]-
                              -> <FACTORIAL>-
                                    -> [NUMBER: *z].
```

The input to the factorial actor is a number whose value is contained within the variable n. The output is also a number and its value will be placed in the variable x when the actor FACTORIAL has completed. The actors in the graphs are <IDENT>, <ADD1>, <MULTIPLY>, <SUB1>, and the recursive function call <FACTORIAL>.

Once actors are defined they may be used within a schema definition. The next example defines a schema for TRAVEL. The actor <MULTIPLY> is contained within the actor graph. It calculates the distance between two points, the starting location and the destination by using two inputs, speed and time-period, to calculate the result.

```
schema for TRAVEL(x) is
   [TRAVEL: *x]-
     (AGNT) -> [PERSON]
     (INST) -> [VEHICLE]
     (RATE) -> [SPEED: @]-
                 -> <MULTIPLY>-
                      <- [TIME-PERIOD: @*z]
                      -> [DISTANCE: @*w]-
                            (SRCE) -> [PLACE: *y],,
     (DUR)  -> [TIME-PERIOD: @*z]
     (DEST) -> [PLACE] <- (DEST) <- [DISTANCE: @*w]
     (SRCE) -> [PLACE: *y].
```

2.4 Database usage for Conceptual Graphs

Before ending the chapter on conceptual graphs, possible uses of conceptual graphs for application in database usage will be presented. Examples and descriptions will be provided to show how the conceptual graphs relate to database work.

In database design the knowledge is stored as a set of database descriptors that describe hundreds or thousands of records. Although this seems to be quite adequate at the present time, database design has been moving toward the representation of highly structured applications. This move has made database design teams aware of the need to add more complex relationships with fewer descriptor items. The knowledge for present database systems consists of the meaning of the knowledge and the rules that are necessary for representing the knowledge in terms of real world situations. A field called database semantics has been studying the meaning of the knowledge and the rules to use it. In Sowa[1980], he listed seven kinds of knowledge necessary for database semantics. Each of these are present in the conceptual graph theory. A summary of each is listed below.

1) Type Hierarchy - Entities are ordered according to their level of generality, such as Collie, Dog, Animal, Living-Thing, Physical-Object, and Entity. Type definitions in the conceptual graph theory are used to represent the type hierarchy.

2) Functional Dependencies - The notation must show which entity types ere keys end which entity types ere functionally dependent on the keys. It must also contain quantifiers thet show whether a function is many-to-one, one-to-one, or n-to-m. In the conceptual graphs actors show how referents of concepts can be found in the detabase.

3) Domain Roles - The notation must describe the role that the dependency represents. Functional dependencies show that concepts cen be related and the domain roles show what that reletionship means.

4) Definitions - Entities, concept end relation types, end ectors must be definable in terms of structures of other concepts. The genus of the concept PERSON would have one definition such as EMPLOYEE and the cheracteristic of EMPLOYEE would be "one who works for a company."

5) Schemata - For eech type of concept or entity, the notation must describe the normally occurring or default roles that it plays with respect to other concepts. Schemata show beckground information, for example a schema for EMPLOYEE could contain an employee number, salery amount, depertment name, etc.

6) Procedural Attachments - The notetion should indicate how en external procedure may be related to a functional dependency and under whet condition it would be invoked to compute the function. In conceptual graphs actors ere bound to schema which show how external procedures can compute the referents of concepts.

7) Inferences - Rules of inference must be included to determine implications thet follow from the explicitly stored data and can detect violations of constraints on the data.

Given the following record layout for a database system, the detebase design would contain e descriptor for each item. The descriptors for the teble are NAME

(name of the owner of the policy), POLICY NUMBER (number of the insurance policy for the person), PAYMENT AMOUNT (amount paid by a person for a policy), and DATE PAID (date the amount wes paid).

NAME	POLICY NUMBER	PAYMENT AMOUNT	DATE PAID
Tim Hines	2286	$250.00	1-16-85
Susan Brick	9248	$345.00	9-11-85
Ted Bess	1065	$275.00	2-19-85
John Sorden	837	$190.00	3-23-84
Debra Bickley	5584	$290.00	6-15-85

The table ebove can be represented by a schema definition with a conceptual greph representing each descriptor field. The concept type for the schema would be PAYMENT. In the scheme the concepts representing the descriptor fields of the table would be [PERSON] representing the NAME, [POLICY-NUMBER] representing the POLICY NUMBER, [MONEY] representing the PAYMENT AMOUNT, and [DATE] which represents the DATE PAID. The concept [INSURER] in the conceptual graph contains a fixed value where the recipient of all the payments are made to the company "Mutual of Omaha."

```
Schema for PAYMENT(x) is
   [PAYMENT: *x]-
       (AGNT) -> [PERSON]
       (CHRC) -> [POLICY-NUMBER]
       (OBJ)  -> [MONEY: @]
       (PTIM) -> [DATE]
       (RCPT) -> [INSURER: Mutual of Omaha].
```

With a schema defined for PAYMENT we can now esk ques-

tions on our database. If we asked "How much money did Tim Hines pay on his policy number 2286?", the following graph would be constructed to represent the question. The "?" symbol in the concept [MONEY] shows where the answer of the question should go.

```
[PAYMENT]-
   (AGNT) -> [PERSON: Tim Hines]
   (CHRC) -> [POLICY-NUMBER: 2286]
   (OBJ)  -> [MONEY: @ ?].
```

The query graph shown above maps to the schema PAYMENT and then searches the database to select a match if one exists. The answer from the query graph would be [MONEY: @$250.00] using the data in the table listed above.

The previous query graph selected only one individual as its answer. Queries that retrieve a set of answers can also be represented by conceptual graphs. The question, "What are the payment dates for Tim Hines and Debra Bickley?" would be represented by the following query graph.

```
[PAYMENT]-
   (AGNT) -> [PERSON: {Tim Hines, Debra Bickley}]
   (PTIM) -> [DATE: {?}].
```

The symbol {?} in the concept [DATE] asks for a "set" of answers for the date paid. The query graph is first mapped to the schema PAYMENT. Once the mapping has completed, searching is done on the database for all

matching instances of the schema query graph. The answer returned using the table listed above would be the following.

```
[PAYMENT]-
    (AGNT) -> [PERSON: {Tim Hines, Debra Bickley}]
    (PTIM) -> [DATE: {1-16-85,6-15-85}].
```

While we have shown a possible use for conceptual graphs, there are several things that make them confusing to a user who is designing conceptual graphs. In his book, Sowa omitted the discussion of how to use the conceptual graphs for a complete system, Sowa[1984]. He presented the theory and the representation, but left the rest up to the knowledge representation researcher to explore. One of the omissions is the changes of states which are necessary in expert system development. In recent work Hartley presented a method to show how the states can change, Hartley[1985]. He did this by defining new actors and then adding them to schema definitions. Although his work extends the conceptual graph theory, his representation has many drawbacks. The simplicity in representing the conceptual graphs does not hold true in his work. The issues described here will be presented again in chapter 4 for further discussion.

The next chapter, chapter 3, describes object-oriented environments and their attributes. In this

discussion similarities between the conceptual graphs and the object-oriented languages will be discussed. Chapter 4 presents a formalism for using conceptual graphs based on the object-oriented environments and their attributes. Examples from Hartley's paper will be presented and changes will be made which will make the representation simpler and easier to understand from the user's perspective.

Object-Oriented Programming

Object-oriented programming has grown in popularity because of its modular programming constructs and because of the techniques available for use with them. In section 1.1 we discussed the Procedural/Declarative controversy of the seventies. From the controversy there came only one agreement. We needed to find some sort of bridge between the two types of knowledge representation. The first research attempt to do so used semantic networks and frames as the knowledge representation method. Several knowledge programming systems (ABE, ART, KEE, Strobe, UNITS, etc.) have emerged from this effort. These systems included extensions and variations on object-oriented programming for creating knowledge based systems in terms of objects.

The first object-oriented programming language was Simula created by Dahl in 1967. The first modern founder of object-oriented programming was the language SmallTalk, designed by Alan Kay at Xerox PARC. SmallTalk borrowed the class concept, a form of data abstraction, from the Simula programming language, SmallTalk[1980]. Over the past decade other object-oriented systems have been written that add objects to the programming language Lisp (these include Flavors, Loops, and Object-Lisp.) As Common Lisp has grown in

popularity, an interest to add objects to it has attracted widespread attention. The Common Lisp Objects Committee was formed to standardize the integration of objects into Common Lisp. CommonLoops is one extansion of Common Lisp for objects. Other faaturas that have mada object-oriented programming popular hava been the artificial intelligence workstations. The workstations environment adapts itself easily to modular programming constructs. They hava graphics packages which are integrated together with tha workstations programming anvironment making for a powarful knowledge anginearing tool. In chapter 3 we will describe the attributes and tha anvironmants of object-orientad programming.

3.1 Objects and Classes

An object is a form of data abstraction. It combines the propertias of procedures and data. The object saves its local state and performs procedures. For exampla, an object might be an AUTOMOBILE. The instance variables (data) in the object might be x-position, y-position, x-velocity, y-velocity, and mass, and tha methods (procedures) could be SPEED and DIRECTION. When the methods SPEED and DIRECTION are executed they can retriave the valuas contained in the object's instance variables. The mathods may also have temporary variables if storage is neadad to hold values when performing their operations.

An object may reside somewhere in a class-subclass taxonomy of objects. This is also called a type lattice. An object in the type lattice may have a superclass or a subclass. Each class will have common features associated with its subclasses and superclasses, and as you go down the type lattice the objects begin to be more specific in terms of an object's type. A hierarchical description of a type lattice is shown in figure 3.1.1.

```
                          Physical Object
                                /|\
                               / | \
                              /  |  \
                             /   |   \
                            /    |    \
                   Vehicle  Computer  Building
                     /|\
                    / | \
                   /  |  \
                  /   |   \
                 /    |    \
        Automobile  Truck  Boat
            /|\
           / | \
          /  |  \
         /   |   \
        /    |    \
   Sedan  Coupe  Station
                  Wagon
```

Figure 3.1.1 A Hierarchical description of a type lattice.

The type lattice shows that the highest object (class) is the object PHYSICAL-OBJECT. The subclasses of PHYSICAL-OBJECT are VEHICLE, COMPUTER, and BUILDING.

The subclasses of VEHICLE are AUTOMOBILE, TRUCK, and BOAT, and the subclasses of AUTOMOBILE are SEDAN, COUPE, and STATION-WAGON. In the type lattice each object may contain its own data, and methods may be attached to perform operations on the object's data values. This taxonomy of objects lends itself to a modular programming environment. Each object will be treated as a black box. Other objects in the type lattice may know what the facilities' external interfaces guarantee, but nothing else.

An object is just a pattern of some entity. The pattern will contain the instance variables that are associated with the object. The methods used for performing operations can then be attached to that object's pattern. In Flavors the object AUTOMOBILE could be defined by the following example.

```
(defflavor AUTOMOBILE (x-position y-position
                       x-velocity y-velocity
                       mass)
                      ())
```

The "defflavor" function in Flavors initializes a pattern for the object AUTOMOBILE. The object AUTOMOBILE contains the instance variables x-position, y-position, x-velocity, y-velocity, and mass. Earlier we said there may be methods in the object which perform some type of operation. The following Flavors code shows two methods, SPEED and DIRECTION, attached to the object

AUTOMOBILE.

```
(defmethod (AUTOMOBILE :SPEED) ()
     (sqrt (+ (^ x-velocity 2)
              (^ y-velocity 2))))

(defmethod (AUTOMOBILE :DIRECTION) ()
     (atan y-velocity x velocity))
```

If the method DIRECTION were executed the arc tangent (atan) of the instance variables y-velocity and x-velocity would be calculated. The x-velocity and y-velocity instance variables can be found within the object AUTOMOBILE. Methods will be discussed further in section 3.5.

3.2 Inheritance

Instead of placing the instance variables x-position, y-position, x-velocity, y-velocity, and mass, and the methods SPEED and DIRECTION in the object AUTOMOBILE, they might placed higher in the type lattice. This will provide efficiency and will reduce redundant code. In figure 3.1.1 the class VEHICLE is a superclass of AUTOMOBILE. It would be better if we were to place the instance variables and methods (from the object AUTOMOBILE, section 3.1) in the object VEHICLE, so that the objects TRUCK and BOAT could inherit the same instance variables and methods. An even better solution would be the addition of another object called MOVING-OBJECT, as the superclass of VEHICLE. The new type lattice with the object MOVING-OBJECT inserted

into the type lattice of figure 3.1.1 is shown in figure 3.2.1. We will now place the instance variables x-position, y-position, x-velocity, y-velocity, and mass, and the methods SPEED and DIRECTION in the object MOVING-OBJECT. The objects METEOR, VEHICLE, AUTOMOBILE, TRUCK, BOAT, SEDAN, COUPE, and STATION-WAGON can now inherit the instance variables and methods in the object MOVING-OBJECT.

```
                          Physical Object
                               /|\
                              / | \
                             /  |  \
                            /   |   \
                           /    |    \
                     Moving  Computer  Building
                     Object
                      / \
                     /   \
                    /     \
                   /       \
              Vehicle    Meteor
                /|\
               / | \
              /  |  \
             /   |   \
            /    |    \
     Automobile  Truck  Boat
        /|\
       / | \
      /  |  \
     /   |   \
    /    |    \
Sedan  Coupe  Station
               Wagon
```

Figure 3.2.1 The class Moving Object was added to type lattice in figure 3.1.1.

The objects MOVING-OBJECT, AUTOMOBILE, and METEOR are defined with Flavors code in figure 3.2.2. The object MOVING-OBJECT has the instance variables x-position, y-position, x-velocity, y-velocity, and mass, and the methods SPEED and DIRECTION. There are new instance variables contained in the objects AUTOMOBILE, VEHICLE, and METEOR.

```
(defflavor MOVING-OBJECT (x-position y-position
                          x-velocity y-velocity
                          mass)
                         (PHYSICAL-OBJECT))

(defmethod (MOVING-OBJECT :SPEED) ()
       (sqrt (+ (^ x-velocity 2)
                (^ y-velocity 2))))

(defmethod (MOVING-OBJECT :DIRECTION) ()
       (atan y-velocity x-velocity))

(defflavor METEOR (percent-iron)
                  (MOVING-OBJECT))

(defflavor VEHICLE (number-of-passengers engine-power type)
                   (MOVING-OBJECT))

(defflavor AUTOMOBILE (color model year)
                      (VEHICLE))
```

Figure 3.2.2 Flavors code for the objects MOVING-OBJECT, METEOR, VEHICLE, and AUTOMOBILE, and the methods SPEED and DIRECTION in the object MOVING-OBJECT.

Subclasses can inherit properties contained within their superclasses. For example, in figure 3.2.2 the object AUTOMOBILE may inherit the instance variables and methods from the objects VEHICLE and MOVING-OBJECT. The object METEOR can only inherit the instance variables and methods from the object MOVING-OBJECT. If

the object METEOR were to execute the method DIRECTION, it would be inherited from the object MOVING-OBJECT since it is not defined within METEOR.

3.3 Instances

Once we have a pattern for an object, unique instances can be created for that object. When an object instance is created the instance variables in the object can be assigned values. For example, to make an instance in Flavors of the object AUTOMOBILE (defined in figure 3.2.2), zero or more instance variables may be assigned a value. Some instance variables may be inherited through its superclasses. The creation of the object AUTOMOBILE is shown below. The new instance name of the object AUTOMOBILE is "MY-CAR".

```
(setq MY-CAR
   (make-instance 'AUTOMOBILE ':x-position 4.0
                              ':y-position 2.0
                              ':mass     1245.0
                              ':number-of-passengers 4
                              ':color    'blue))
```

The unique instance "MY-CAR" is created with some of its instance variables assigned to values and others that have been left undefined. Some object-oriented languages provide tools to help develop a system. In Flavors there is a function to describe the contents of an object instance. The following example shows the Flavors function "describe" applied to the object "MY-

CAR."

(describe MY-CAR)

```
An object of type AUTOMOBILE bas instance variables:
x-position            4.0
y-position            2.0
x-velocity            unbound
y-velocity            unbound
mass                  1245.0
number-of-passengers  4
engine-power          unbound
type                  unbound
color                 blua
model                 unbound
year                  unbound

Tbe  objact AUTOMOBILE bas the suparclass VEHICLE.
VEHICLE bas tbe superclass MOVING-OBJECT.
```

Aftar an instance of an object is craatad tbe instance variables tbat bave been left undefined or that bave bave been assigned a valua wben created, may be assignad a new value. For axample, assign a value of "1986" could be assigned to the instance variable year and/or a change could be made to the value of tha instance variabla mass.

3.4 Defaults

Default values may be used to initialize values for instance variablas. The default values are used when a new instanca of an object is created. For example, in tbe object AUTOMOBILE it may have default values for tha instance variables modal and year. Tbe following exampla shows bow defaults are used in Flavors.

```
(defflavor AUTOMOBILE ( color
                        (model 'corvette)
                        (year '1986))
                       (VEHICLE))
```

Wben a new instance of the object AUTOMOBILE is created, tbe instance variables model and year will automatically be assigned tbe values "corvette" and "1986", respectively. The next example shows tbe new object instance, "YOUR-CAR", using the above AUTOMOBILE object pattern with default values. Tbe output from the Flavors describe function is also shown below.

```
(setq YOUR-CAR
    (make-instance 'AUTOMOBILE ':x-position 0.0
                               ':y-position 2.0
                               ':color     'red))

(describe YOUR-CAR)

An object of type AUTOMOBILE has instance variables:
x-position           0.0
y-position           2.0
x-velocity           unbound
y-velocity           unbound
mass                 unbound
number-of-passengers unbound
engine-power         unbound
type                 unbound
color                red
model                corvette
year                 1986

Tbe object AUTOMOBILE has the superclass VEHICLE.
VEHICLE bas tbe superclass MOVING-OBJECT.
```

3.5 Methods and Message Passing

Methods can be used to perform operations on the values of an object's instance variables. After a new

instance of an object is created, methods may be executed to perform operations on its instance variables. These may be contained within the instance or they may be inherited from its superclasses. For example, lets create the new instance "TOY-CAR" using the object AUTOMOBILE. The instance "TOY-CAR" can now inherit the methods SPEED and DIRECTION from the object MOVING-OBJECT. The following Flavors code shows the new instance "TOY-CAR" being created using the object AUTOMOBILE.

```
(setq TOY-CAR
     (make-instance 'AUTOMOBILE ':x-position 0.0
                                ':y-position 2.0
                                ':x-velocity 4.0
                                ':y-velocity 3.5
                                ':mass       1.5
                                ':color      'brown))
```

To perform a method a message must be sent to the instance of an object. A message is sent to an object and the object will return a result. This is send and receive message passing in object-oriented programming. In most object-oriented languages it is the only way to perform communication. If we wanted the method DIRECTION to execute for the object "TOY-CAR", a message must be sent to the object "TOY-CAR" telling it to perform the method DIRECTION. The example below shows how a message is sent in Flavors to the instance object "TOY-CAR".

(send TOY-CAR ':DIRECTION)

The object "TOY-CAR" receives the message which says to perform the method DIRECTION. If the object has the method DIRECTION in it or in a superclass, it will execute. If it does not exist then a message would be returned saying that it could not execute the method DIRECTION. In the send example above, the object "TOY-CAR" receives the message to execute the method DIRECTION. The object "TOY-CAR" will then inherit the method DIRECTION from the object MOVING-OBJECT. The method DIRECTION will now calculate the arc tangent of the y-velocity and x-velocity. The instance variables x-velocity and y-velocity in the object "TOY-CAR" have the values 4.0 and 3.5, respectively. The calculation in the method DIRECTION would be (atan 3.5 4.0) or the answer being 0.7188. The message returned to the caller is the result calculated from the method DIRECTION. The value 0.7188 is returned to the caller.

3.6 Daemons and Procedural Attachements

Daemons are used differently in object-oriented languages. Daemons are-- 1) methods that may be invoked before or after a method is executed (called before and after daemons in Flavors); 2) called when an instance variable's value changes or is accessed (active values in Loops); or 3) activated when a value is needed, created, or removed (more commonly known as procedural

attachments.) The following example shows how the before and after daemons are used in Flavors. The ":before" daemon is invoked before the method DIRECTION is performed, and the ":after" daemon is invoked after the method DIRECTION finishes its operations.

```
(defmethod (AUTOMOBILE :before :DIRECTION)
    (print "I am moving in the direction--"))

(defmethod (AUTOMOBILE :DIRECTION)
    (atan y-velocity x-velocity))

(defmethod (AUTOMOBILE :after :DIRECTION)
    (print "The automobile went thataway"))
```

When the above method DIRECTION is invoked the "before" daemon will fire before the execution of the method DIRECTION. When the "before" daemon is executed it prints the message "I am moving in the direction--." After the "before" daemon has finished the method DIRECTION will be executed. In the method DIRECTION it calculates the arc tangent of "x-velocity" and "y-velocity." After the method DIRECTION has executed the "after" daemon will fire. The "after" daemon prints the message "The automobile went thataway." When the "after" daemon finishes its execution, the result returned from the invocation of the DIRECTION method will be the last result computed in the DIRECTION method. The result returned from the invocation of the DIRECTION method will be the result computed from the arc tangent of "x-velocity" and "y-velocity."

3.7 Other Object-Oriented Language Syntax

In the previous sections of chapter 3, the syntax of Flavors was used to present the constructs and techniques available in object-oriented programming languages. In this section we will present the syntax from other object-oriented programming languages. These include SmallTalk, Loops, and CommonLoops. Also shown is an example from the knowledge programming system KEE.

SMALLTALK

Figure 3.7.1 shows a SmallTalk class template of the class DepositRecord. It has a superclass called Object and two instance variables, "date" and "amount." The rest of the template describes the methods that may be used to perform operations using the values of the instance variables. Within the class, messages may be sent to other instances of classes or to itself. When a message is received by a class from another class, a method is performed using the values of its own instance variables.

```
+------------------+-----------------------+
|class name        | DepositRecord         |
+------------------+-----------------------+
|superclass        | Object                |
+------------------+-----------------------+
|instance variables | date  amount         |
+------------------+-----------------------+
|methods           |                       |
+------------------+                       |
|    of:depositAmount on: depositDate | |  |
|                                          |
|       date <-- depostDate.               |
|       amount <-- depostAmount            |
|                                          |
|    amount| |                             |
|       ^ amount                           |
|                                          |
|    balanceChange| |                      |
|       ^ amount                           |
+------------------------------------------+
```

Figure 3.7.1 A SmallTalk class definition for the class DepositRecord.

LOOPS

In figure 3.7.2, an example of a Loops class definition called TRUCK is presented. Tbe class TRUCK bas superclasses VEHICLE and CARGOCARRIER. The class contains class variables that are shared by all instances of the class along with instance variables tbat are specifio to a particular class instance. The class also contains methods tbat may be performed on a specifio instance of tbe class TRUCK.

```
TRUCK

MetaClass Class
    EditedBy (*dgh "29-Feb-85 4:32)
    doc(** This sample class illustrates
           the syntax of classes in Loops)

Supers (VEHICLE CARGOCARRIER)

ClassVariables
    tankCapacity 79 doc(*gallons of diesel)

InstanceVariables
    owner          PIE doc(* Owner of the truck)
    highway         66 doc(* Route number of the
                             highway)
    milePost         0 doc(* Location on the highway)
    direction     East doc(* One of North, East, South,
                             or West)
    cargoList       NL doc(* List of cargo descriptions)
    totalWeight      0 doc(* Current weight of cargo
                             in tons)

Methods
    Drive   Truck.Drive   doc(* Moves the vehicle in
                                the simulation)
    Park    Truck.Park    doc(* Parks the truck in a
                                double space)
    Display Truck.Display doc(* Draws the truck in
                                the display)
```

Figure 3.7.2 Loops class definition for the class TRUCK

COMMONLOOPS

CommonLoops is similar to Flavors in the way object-oriented programming is written. CommonLoops uses CommonLisp to represent object-oriented constructs similar to the way Flavors is implemented using ZetaLisp. The CommonLisp function "defstruct" has been extended for CommonLoops to allow for the definition of class templates. This is similar to the Flavors "defflavor" function for defining classes. Methods in Com-

monLoops are defined using the function "defmethod". This is the same function used for defining methods in Flavors. The designers of CommonLoops have considered the extension of the CommonLisp function "defun" to allow for the definition of methods thus eliminating the "defmethod" function in CommonLoops. An example of CommonLoops is shown in figure 3.7.3. The example defines the class TITLED-WINDOW and a method called INSIDEP. The class TITLED-WINDOW includes the superclasses WINDOW and TITLED-THING. The method INSIDEP determines whether or not a point is located inside a given window.

```
(defstruct (TITLED-WINDOW
             (:include (WINDOW TITLED-THING))))

(defmethod INSIDEP ((w WINDOW) (x INTEGER) (y INTEGER))
    ; code for determining if the point x y is inside
      the window would go here
    ...)
```

Figure 3.7.3 A CommonLoops class definition of TITLED-WINDOW and the definition of the method INSIDEP.

KEE

An example from the knowledge programming system KEE is shown in figure 3.7.4. It shows a KEE frame definition of the class TRUCKS and procedural information which may be performed on an instance of the class TRUCKS. The frame shows two attributes, HEIGHT and WEIGHT, of the class TRUCKS. The frame TRUCKS has

procedural information to DIAGNOSE a TRUCKS electrical faults. It also contains a slot called ELECTRICAL.FAULTS which holds known electrical faults found from the DIAGNOSE function.

```
------------------------------------------------
 Frame: TRUCKS in knowledge base TRANSPORTATION
     Superclasses: VEHICLES
     Subclasses: BIG.NON.RED.TRUCKS,
                 HUGE.GRAY.TRUCKS
     MemberOf: CLASSES.OF.PHYSICAL.OBJECTS
------------------------------------------------
 MemberSlot: HEIGHT from PHYSICAL.OBJECTS
     ValueClass: INTEGER
     Cardinality.Min: 1
     Cardinality.Max: 1
     Units: INCHES
     Comment: "Height in inches."
     Values: Unknown

 MemberSlot: LENGTH from PHYSICAL.OBJECTS
     ValueClass: NUMBER
     Units: METERS
     Comment: "Length in meters."
     Values: Unknown
------------------------------------------------
 Unit: TRUCKS in knowledge base TRANSPORTATION
     Superclasses: VEHICLES
     Subclasses: BIG.NON.RED.TRUCKS,
                 HUGE.GRAY.TRUCKS
     MemberOf: CLASSES.OF.PHYSICAL.OBJECTS
------------------------------------------------
 MemberSlot: DIAGNOSE from TRUCKS
     Inheritance: METHOD
     ValueClass: METHODS
     Cardinality.Min: 1
     Cardinality.Max: 1
     Comment: "A method for diagnosing electrical faults."
     Values: TRUCK.DIAGNOSIS.FUNCTION

 MemberSlot: ELECTRICAL.FAULTS from TRUCKS
     Comment: "Faults found by the DIAGNOSIS method."
     Values: Unknown

 MemberSlot: LOCATION from PHYSICAL.OBJECTS
     Values: Unknown
     ActiveValues: UPDATE.LOCATION
```

Figure 3.7.4 KEE frame definition and procedural information for the class TRUCKS.

The following chapter will present techniques for using Sowa's conceptual graphs as the knowledge representation method in an objeot-oriented programming environment. Section 3.7 provided a look at several object-oriented programming languages none of which provide a high-level representation method needed to represent the kind of knowledge necessary for expert system development. The need for a high-level representation method was discussed in chapter 1. Using conceptual graphs in an object-oriented language it will provide the user with a high-level knowledge representation method and modular constructs and techniques to use them.

4

Conceptual Object-Oriented Programming

John Sowa's conceptual graph theory has been chosen as the knowledge representation method for representing both declarative and procedural knowledge while applying object-oriented constructs and techniques. Chapter 4 uses the same constructs and techniques earlier described in Chapter 3 for object-oriented languages with one exception. The Flavors examples in Chapter 3 are replaced with conceptual graph examples in Chapter 4.

The techniques and constructs using conceptual graphs for object-oriented programming will be discussed using a partial design from the expert system CRIB, Hartley[1985]. When techniques and constructs are not easily shown by using the expert system CRIB, smaller examples unrelated to CRIB will be presented instead. The last example in this chapter presents one perspective of how the reader-writer problem could be solved using conceptual graphs and applying object-oriented constructs and techniques to them.

4.1 Type Hierarchy

To allow for a modular system design tool, "type" graphs will be used to perform the initial design of a system where only the objects and classes are defined.

The technique used here will enable the user to define the objects in a system and each object's super and sub objects. This definition of "type" graphs provides the user with a tool in developing conceptual object-oriented systems without defining the specifics of how the objects are used. This tool can be described as a software engineering tool to help the user clearly define a system before actually implementing it. For example, using the type hierarchy in Figure 4.1.1, each node describing some object or class of objects can be defined by a "type" graph. These will be simple graphs containing only those concepts that represents the object's immediate super objects. In Figure 4.1.2 some of the objects in the type hierarchy of Figure 4.1.1 have been defined with "type" graphs.

There are two objects shown in Figure 4.1.1 that are part of the bare system. These are OBJECT and ABSURD. The object OBJECT is the highest in the type hierarchy and any system being defined will come directly under it or under other objects contained in the type hierarchy. The lowest object is ABSURD. All objects that do not contain any other part will automatically contain the object ABSURD. This encloses any system being built to make it a closed system. The "type" graphs in Figure 4.1.2 represent part of the CRIB expert system. Section 4.2 will describe how the CRIB expert system tries to locate a faulty field

replaceable unit in the computer hardware.

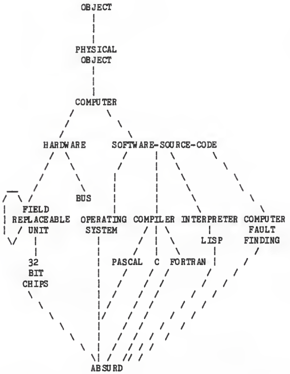

Figure 4.1.1 Type hierarchy showing objects and classes.

```
Type Object(x) is [].
Type Physical-Object(x) is [OBJECT].
Type Computer(x) is [PHYSICAL-OBJECT].
Type Hardware(x) is [COMPUTER].
Type Software-Source-Code(x) is [COMPUTER].
Type Field-Replaceable-Unit(x) is [HARDWARE].
Type 32-Bit-Chip(x) is [FIELD-REPLACEABLE-UNIT]
```

Figure 4.1.2 Type graphs defining objects and classes from Figure 4.1.1.

The first type graph in Figure 4.1.2 is the definition of the type "Object". Notice that the concept within the type definition is blank, and shows that there are no other types higher in the hierarchy. This is similar to Loops and Smalltalk programming where the top node in the programming environment is the class "Object." It is also similar to Flavors where the top flavor or object is "Vanilla". Each type graph defines itself in terms of a class or object one level higher in the type hierarchy. In Figure 4.1.2, the class "Computer" is defined in terms of the "type" graph "Physical-Object" and the type definition of "Hardware" is defined in terms of the class "Computer."

Multiple super objects can be defined using the "type" graphs. Figure 4.1.3 shows a type hierarchy that contains an object that is defined in terms of two super objects.

```
                    OBJECT
                      |
                      |
                    PERSON
                    /    \
                   /      \
                  /        \
            FACULTY     STUDENT
           /   |   \       |      \
          /    |    \      |       \
         /     |     |     |        \
 VISITING  REGULAR | UNDERGRADUATE GRADUATE
 FACULTY   FACULTY |    STUDENT      STUDENT
                    \                 /
                     \               /
                        GRADUATE
                        TEACHING
                        ASSISTANT
```

Figure 4.1.3 Type hierarchy showing a class with multiple super class definitions.

The type definitions in Figure 4.1.4 show how the class "Graduate-Teaching-Assistant" is defined in terms of multiple objects. The type definition of "Graduate-Teaching-Assistant" defines itself in terms of the object "Graduate-Student" and is also a member (MMBR) of the "Faculty". The other type definitions in Figure 4.1.3 are single type definitions defining objects with one super object.

Type Person(x) is [OBJECT: *x].

Type Student(x) is [PERSON: *x].

Type Faculty(x) is [PERSON: *x].

Type Graduate-Student(x) is [STUDENT: *x].

Type Graduate-Teaching-Assistant(x) is
    [GRADUATE-STUDENT: *x] -> (MMBR) -> [FACULTY].

Figure 4.1.4 Type graph definitions showing multiple type definitions using the type hierarchy from Figure 4.1.3.

On a computer that contained a powerful graphics package, the "type" graphs could be used to quickly draw out a picture of a type hierarchy similar to the one shown in Figures 4.1.1 and 4.1.3. The next section describes how to define an object's attributes. At that time the method(s), if any, can be denoted, see section 4.5.

4.2 Objects and Classes

The "schema" graphs will be used to describe an object's attributes. The attributes of an object are its super objects, parts that are other objects, characteristics about the object, and the methods that can be performed on the object. The "schema" graphs shown in Figure 4.2.1 describe objects that are contained within the expert system CRIB. An explanation of how CRIB works is given next.

The strategy for CRIB is to "divide and conquer".

Diagnostic tests are performed to discover which parts of the machine are assumed to be faulty and which parts are assumed to be non-faulty. Using this scheme a faulty field replaceable unit is searched for. The hardware of the machine can be thought of as a hierarchy of field replaceable units where each field replaceable unit may be made of zero or more field replaceable units. The hierarchy of the machine hardware is given and will remain fixed throughout the diagnosis of the computer. Knowledge about past fault-finding and their invalid symptoms will also be kept. The following steps describe how the machine could be diagnosed in the attempt to locate a faulty field replaceable unit.

1) The initial start sends a message to the hardware asking it to perform the method "diagnose".

2) The "diagnose" method sends a message to the field replaceable unit asking it to perform the method "choose". The "choose" method chooses a test which takes the least amount of time to perform from the current field replaceable unit's sub field replaceable units. The "choose" method returns the test chosen to the "diagnose" method.

3) The "diagnose" method sends a message to the field replaceable unit to perform the test chosen above in step 1. The "perform" method observes the symptoms from the field replaceable unit and returns those symptoms to the "diagnose" method. In "diagnose" the symptoms found are added to the current symptoms found from previous tests.

4) The "diagnose" method sends a message to the field replaceable unit to analyse its current symptoms. The "analyse" method tries to determine if the field replaceable unit is faulty.

If it is, the field replaceable unit is returned to the "diagnose" method as a faulty unit.

5) The "diagnose" method halts if a faulty field replaceable unit has been found. Otherwise a message is sent to the hardware asking it to perform the same "diagnose" method again on a different division of the hardware.

The "type" graphs defined in Figure 4.1.2 are further defined by "schema" graphs in Figure 4.2.1. The objects COMPUTER, HARDWARE, and FIELD-REPLACEABLE-UNIT are defined with "schema" graphs.

```
Schema for Computer(x) is
  [COMPUTER: *x]-
     (SUPER) -> [PHYSICAL-OBJECT]
     (PART)  -> [HARDWARE: {*}]
     (PART)  -> [SOFTWARE: {*}]
     (CHRC)  -> [SERIAL-NUMBER]
     (MTHD)  -> [OPERATING-SYSTEM-COMMANDS]
     (MTHD)  -> [COMPILERS]
     (MTHD)  -> [INTERPRETERS].

Schema for Hardware(x) is
  [HARDWARE: *x]-
     (SUPER) -> [COMPUTER]
     (PART)  -> [FIELD-REPLACEABLE-UNIT]-
                 (OBJ) -> [TEST: RESP{*}]-
                            (DUR) -> [TIME: {*}],,
     (PART)  -> [BUS: {*}]
     (ATTR)  -> [CURRENT-SYMPTOMS: {*}]
     (MTHD)  -> [COMPUTER-FAULT-FINDING].

Schema for Field-Replaceable-Unit(x) is
  [FIELD-REPLACEABLE-UNIT: *x]-
     (SUPER) -> [HARDWARE]
     (PART)  -> [FIELD-REPLACEABLE-UNIT: RESP{*}]-
                 (OBJ) -> [TEST: RESP{*}]-
                            (DUR) -> [TIME: {*}],,
     (PART)  -> [32-BIT-CHIP: {*}]
     (OBJ)   -> [INVALID-SYMPTOMS: {*}]
     (ATTR)  -> [CURRENT-SYMPTOMS: {*}]
     (MTHD)  -> [FIELD-REPLACEABLE-UNIT-FAULT-FINDING].
```

Figure 4.2.1 Schema graphs describing the objects computer, hardware, and field-replaceable unit.

In Figure 4.2.1 the schema graph for "Computer" says that it has a (SUPER) object called [PHYSICAL-OBJECT], two parts, one being the (PART) [HARDWARE] and the other being the (PART) [SOFTWARE]. The "Computer" has the characteristic (CHRC) [SERIAL-NUMBER]. The last part of the "schema" graph shows the methods that can be called upon and used by the object "Computer". The methods are [OPERATING-SYSTEM-COMMANDS], [COMPILER] (e.g., Fortran, C, etc.), and [INTERPRETERS] (e.g. Lisp). The other two "schema" graphs "Hardware" and "Field-Replaceahle-Unit" are similar to the "schema" graph of the object "Computer". The "schema" graph for the object "Hardware" has the entire set of [FIELD-REPLACEABLE-UNITS] with respective [INVALID-SYMPTOMS] that have been previously found in prior computer fault finding tests. Each of the [FIELD-REPLACEABLE-UNITS] contains the set of tests with respective durations (DUR) of how long it takes to perform each test. The "schema" graph for the object "Field-Replaceable-Unit" will have its own set of known [TESTS] and the duration (DUR) of how long it takes to perform that test. The completion of the two "schema" graphs will not be further discussed.

4.3 Inheritance

Each object that has has a (SUPER) object(s) may inherit attributes and methods from that (SUPER)

object(s). For example, the object "Hardware", defined by the "schema" graph in Figure 4.2.1, may inherit the methods and attributes, except for the (PART) objects, contained in the object "Computer". For instance, "Hardware" can use the [OPERATING-SYSTEM-COMMANDS], it can also access the [SERIAL-NUMBER] of the "Computer". The attributes of the "Computer's" (SUPER) object, [PHYSICAL-OBJECT] can also be inherited by the object "Hardware".

4.4 Instances

Each object defined by "schema" graphs may have unique instances. For example, using the schema graphs in Figure 4.2.1, there may be many computers that can be made, and each computer may have several pieces of hardware attached to it. Each piece of hardware contained in the computer may also have many field replaceable units. Figure 4.4.1 shows how a unique object can be made using an actor called <MAKE-OBJECT>.

```
<MAKE-OBJECT>-
    <- [OBJECT]
    -> [OBJECT: *loc].

<MAKE-OBJECT>-
    <- [COMPUTER: Xerox-1186-1]-
          (CHRC) -> [SERIAL-NUMBER: 4692AHSN]
          (PART) -> [HARDWARE: Xerox-1186-1]
    -> [COMPUTER: *loc= "Xerox-1186-1"= #74902561].
```

Figure 4.4.1 Making a unique object of type "Computer" described by the schema graph in Figure 4.2.1.

The object being produced from Figure 4.4.1 is the object "Computer". The input to the actor <MAKE-OBJECT> is an object name along with any attributes thet will be set upon the creetion of the object. The value returned is the name of the object produced along with an 8-bit address identifying the unique object and steting its location in memory. When the actor <MAKE-OBJECT> makes a unique object for e scheme graph, an individuel graph is produced end kept in memory. Figure 4.4.2 shows the unique individual graph for the object "Computer" that wes produced from the <MAKE-OBJECT> in Figure 4.4.1. The individuel graphs for "Hardware" and two "Field-Repleceehle-Units" are also presented.

```
Individual Computer(Xerox-1186-1) is
  [COMPUTER: Xerox-1186-1]-
    (CHRC) -> [SERIAL-NUMBER: 4692AHSN]
    (PART) -> [HARDWARE: Xerox-1186-1].

Individual Hardware(Xerox-1186-1) is
  [HARDWARE: Xerox-1186-1]-
   (ATTR) -> [CURRENT-SYMPTOMS: {}]
   (PART) -> [FIELD-REPLACEABLE-UNIT: Xerox-1186-1-1A]-
               (OBJ) -> [TEST: RESP{D,E}]-
                           (DUR) -> [TIME: {3,8}ms].

Individual Field-Replaceable-Unit(Xerox-1186-1-1A) is
  [FIELD-REPLACEABLE-UNIT: Xerox-1186-1-1A]-
    (OBJ)  -> [TEST: RESP{D,E}]-
                (DUR) -> [TIME: {3,8}ms],
    (ATTR) -> [CURRENT-SYMPTOMS: {}].
    (OBJ)  -> [INVALID-SYMPTOMS: {A,P}]
    (PART) -> [FIELD-REPLACEABLE-UNIT: Xerox-1186-1-1B]
                (OBJ) -> [TEST: RESP{C}]-
                           (DUR) -> [TIME: {5}ms].

Individual Field-Replaceable-Unit(Xerox-1186-1-1B) is
  [FIELD-REPLACEABLE-UNIT: Xerox-1186-1-1B]-
    (OBJ)  -> [TEST: RESP{C}] -> (DUR) -> [TIME: {5}ms]
    (ATTR) -> [CURRENT-SYMPTOMS: {}].
    (OBJ)  -> [INVALID-SYMPTOMS: {K}].
```

Figure 4.4.2 Individual graphs of objects or classes using the schema graph definitions of Figure 4.2.1.

To aid in the debugging of an object-oriented system that is currently being developed, an actor called <DESCRIBE-OBJECT> can be used to retrieve an individual type graph and its attributes. Figure 4.4.3 shows how the actor <DESCRIBE-OBJECT> can be used to print the description of the individual graph of the object "Computer" with the unique object being "Xerox-1186-1".

```
<DESCRIBE-OBJECT> <- [COMPUTER: Xerox-1186-1].

[COMPUTER: Xerox-1186-1]-
    (SUPER) -> [PHYSICAL-OBJECT]
    (PART) -> [HARDWARE: Xerox-1186-1]
    (PART) -> [SOFTWARE]
    (CHRC) -> [SERIAL-NUMBER: 4692AHSN]
    (MTHD) -> [OPERATING-SYSTEM-COMMANDS]
    (MTHD) -> [COMPILERS]
    (MTHD) -> [INTERPRETERS].
```

Figure 4.4.3 The actor describe-object being used to print a unique object of type "Computer".

4.5 Methods and Message Passing

The four steps described in section 4.2 for the expert system CRIB are defined by the method graphs in Figure 4.5.1. These are Diagnose, Choose, Perform, and Analyse. To define methods the keyword "Method" is used to represent method graphs. The method may also have the option to define a set of methods denoted by the keyword "class" after the name "Method". In Figure 4.5.1 the method class for "Computer-Fault-Finding" contains one method that may be called, which is DIAGNOSE. There is also a method class defined for the three methods CHOOSE, PERFORM, and ANALYSE. It is the method class "Field-Replaceable-Unit-Fault-Finding". The definition of the method class "Computer-Fault-Finding" will not do any operations it is merely a place holder for other methods or method classes.

```
Method class for Computer-Fault-Finding(x) is
  [COMPUTER-FAULT-FINDING: *x]-
    (MTHD) -> [DIAGNOSE].

Method class for Field-Replaceable-Unit-
                              Fault-Finding(x) is
  [FIELD-REPLACEABLE-UNIT-FAULT-FINDING: *x]-
    (MTHD) -> [CHOOSE]
    (MTHD) -> [PERFORM]
    (MTHD) -> [ANALYSE].
```

Figure 4.5.1 Defines a class of methods for computer fault finding and for field replaceable unit fault finding.

The method "Diagnose", in Figure 4.5.2, sends a message to the current [FIELD-REPLACEABLE-UNIT] asking it to perform the method "Choose". The message passing is denoted by the actor <MSGC> which stands for "message controller". If the method "choose" finds a [TEST] to "Perform" on a [FIELD-REPLACEABLE-UNIT] the test is returned to the "Diagnose" method. The next step is to send a message to the [FIELD-REPLACEABLE-UNIT] asking it to perform the method "Perform" using the [TEST] that has been sent to it. The value returned from the "Perform" method is the [OBSERVABLE-SYMPTOMS] which are added to the [CURRENT-SYMPTOMS] using the actor <ADD-SETS>. The last step sends a message the [FIELD-REPLACEABLE-UNIT] which had the [TEST] performed on it asking it to "Analyse" itself. If the "Analyse" method returns a "faulty" [FIELD-REPLACEABLE-UNIT] the "Diagnose" method "<HALTs>" and the "faulty" [FIELD-REPLACEABLE-UNIT] is returned to its caller. Otherwise a message is sent to itself (the hardware) asking it to

perform the same "Diagnose" method.

```
Method for Diagnose(x) is
  [DIAGNOSE: *x]-
    (CLASS) -> [HARDWARE]
    (MTHDCLASS) -> [COMPUTER-FAULT-FINDING]
    <MSGC>-
      <- [FIELD-REPLACEABLE-UNIT] -> (MTHD) -> [CHOOSE]
      -> [TEST: *ts]-
            (OBJ) <- [FIELD-REPLACEABLE-UNIT: *fru]-
    <MSGC>-
      <- [FIELD-REPLACEABLE-UNIT: *fru]-
            (MTHD) -> [PERFORM]
            (OBJ)  -> [TEST: *ts],
      -> [OBSERVABLE-SYMPTOMS: *os={*}]-
           <ADD-SETS>-
              <- [CURRENT-SYMPTOMS: *cs={*}]
              -> [CURRENT-SYMPTOMS: *cs]-
    <MSGC>-
      <- [FIELD-REPLACEABLE-UNIT: *fru]-
           (MTHD) -> [ANALYSE],
      -> [FIELD-REPLACEABLE-UNIT: *fru]-
           (ATTR) -> [FAULTY] -> <HALT>,
      -> [DIAGNOSE]-
           (MTHD) <- [HARDWARE: *self] -> <MSGC>,
    (RSLT) -> [FIELD-REPLACEABLE-UNIT: *fru]-
                (ATTR) -> [FAULTY].
```

Figure 4.5.2 Method graph for the CRIB "diagnose" method.

When the method "Choose", shown in Figure 4.5.3, is called it chooses the [TEST] that discriminates best between its sub [FIELD-REPLACEABLE-UNIT's] [TESTs]. The manner in which it chooses a test is done using the actor <SRCH3>. The search actor performs a parallel search searching on a set of [TESTs] which might yield [INVALID-SYMPTOMS] in a partially matched set, and on a set of respective [TIMEs] relating to the corresponding partially matched [TESTs]. The two sets are searched while matching them against the minimum <MIN> amount of

[TIME] for the partially matched [TESTs]. When a [TEST] has been found it is returned to the caller which here is the "Diagnose" method.

```
Method for Choose(x) is
 [CHOOSE: *x]-
  (CLASS)->[FIELD-REPLACEABLE-UNIT]
  (MTHDCLASS)->[FIELD-REPLACEABLE-UNIT-FAULT-FINDING]
  (OBJ)->[TEST: *t]-
    (INST)<-[DISCRIMINATE]-
            (MANR)->[BEST]
            (OBJ)->[FIELD-REPLACEABLE-UNIT: RESP{*}]-
                  (PART)<-[FIELD-REPLACEABLE-UNIT]
                  (ATTR)->[INVALID-SYMPTOMS: *is={*}],,
    (DUR) -> [TIME: *tm],
  <SRCH3>-
    <-[TEST: *ts=RESP{*}]-
       (CHRC)->[INVALID-SYMPTOMS: RESP{*}]-
                <-<DIFF>-
                    <-[INVALID-SYMPTOMS: {*}]-
                        <-<JOIN>-
                            <-[INVALID-SYMPTOMS: *is],,
                    <- [CURRENT-SYMPTOM: {*}],,
       (DUR) -> [TIME: {*}],
    <-[TIME: RESP{*}]-
         (DUR) <- [TEST: *ts]
         -> <MIN> -> [TIME: *tm],
    <-[TIME: *tm]
    ->[TEST: *t],
  (RSLT)->[TEST: *t]-
              (OBJ)<-[FIELD-REPLACEABLE-UNIT].
```

Figure 4.5.3 Definition of the method "Choose"

The method "Perform" in Figure 4.5.4 performs a [TEST] on a [FIELD-REPLACEABLE-UNIT]. The [OBSERVED-SYMPTOMS] and the [TEST] are printed <PRINT> to the user. The result returned to the caller is the [OBSERVED-SYMPTOMS].

```
Method for Perform(x) is
  [PERFORM: *x]-
    (CLASS)->[FIELD-REPLACEABLE-UNIT]
    (OBJ)->[TEST]-
            -> <PRINT> <- [OBSERVED-SYMPTOMS: *os={*}]
    (RSLT)->[OBSERVED-SYMPTOMS: *os].
```

Figure 4.5.4 Definition of the method "Perform".

The method "Analyse", shown in Figure 4.5.5 searches through the set of [CURRENT-SYMPTOMS] trying to make a match against the [INVALID-SYMPTOMS] known from previous computer fault finding. If a "faulty" [FIELD-REPLACEABLE-UNIT] is found then it is returned to the "Diagnose" method.

```
Method for Analyse(x) is
 [ANALYSE: *x]-
   (CLASS) -> [FIELD-REPLACEABLE-UNIT]
   (MTHDCLASS)->[FIELD-REPLACEABLE-UNIT-FAULT-FINDING]
   (CHRC)->[INVALID-SYMPTOMS: {*}]-
           (CHRC)<-[FIELD-REPLACEABLE-UNIT: *f]
           -><SRCH2>-
              <-[CURRENT-SYMPTOMS: {*}]-
                 (CHRC)<-[FIELD-REPLACEABLE-UNIT: *f]
              -> [FIELD-REPLACEABLE-UNIT: *f]-
                    (ATTR) -> [FAULTY],,,
   (RSLT) -> [FIELD-REPLACEABLE-UNIT: *f]-
                (ATTR) -> [FAULTY].
```

Figure 4.5.5 Definition of the method "Analyse".

The initial start up of the expert system CRIB, shown in Figure 4.5.6, would be performed by sending a message to the [HARDWARE] asking it to perform the method "Diagnose". The value returned from the method would be a "faulty" [FIELD-REPLACEABLE-UNIT] if one is found.

```
<MSGC>-
   <- [HARDWARE: Xerox-1186-1] -> (MTHD) -> [DIAGNOSE]
   -> [FIELD-REPLACEABLE-UNIT: ?] -> (ATTR) -> [FAULTY].
```

Figure 4.5.6 Sending a message to a computer asking it to perform the method "Diagnose" and returning a "faulty" field replaceable unit if one is found.

Messages can be denoted in different ways. Figure 4.5.7 shows some of the possible formations of how a message can be sent.

```
<MSGC> <- [METHOD]
       (a)

[METHOD: 1] -> <MSGC> -> [METHOD: 2] -> <MSGC>.
       (b)

[METHOD: 1] -> <MSGC> -
                 -> [METHOD: 2] -> <MSGC>,
                 -> [METHOD: n] -> <MSGC>.
       (c)

<MSGC>-
   <- [METHOD: 1]
   <- [METHOD: n]
   -> [METHOD: n+1] -> <MSGC>.
       (d)

<MSGC> -
   <- [METHOD: 1]
   <- [METHOD: n]
   -> [METHOD: n+1] -> <MSGC>,
   -> [METHOD: m]   -> <MSGC>.
       (e)
```

Figure 4.5.7 Notation for message passing where multiple methods can fire multiple methods.
(a) A Single method being invoked.
(b) Single method invoking a single method.
(c) Single method invoking multiple methods.
(d) Multiple methods invoking a single method.
(e) Multiple methods invoking multiple methods.

It is possible to set up defaults within concepts and section 4.6 describes how defaults can be used in

conceptual graphs.

4.6 Defaults and Conditions

Defaults may be used within the schema and method graphs. In the schema graph shown in Figure 4.6.1, the object "Hardware" has defaults set up in the (PART) [FIELD-REPLACEABLE-UNIT], the [INVALID-SYMPTOMS] for each [FIELD-REPLACEABLE-UNIT], and the set of [TESTs] with respective durations (DUR) of [TIME] that it takes to perform each [TEST]. In the method graph the default is the [TEST: C]. If, when the method "Perform" is called, there is no test to perform it will perform the test "C" and return the observable symptoms.

```
Schema for Hardware(x) is
 [HARDWARE: *x]-
   (SUPER) -> [COMPUTER]
   (PART) -> [FIELD-REPLACEABLE-UNIT: Xerox-1186-1-1A]-
               (OBJ) -> [TEST: RESP{D,E}]-
                              (DUR) -> [TIME: {3,8}],,
   (PART)  -> [BUS: {*}]
   (ATTR)  -> [CURRENT-SYMPTOMS: {*}]
   (MTHD)  -> [COMPUTER-FAULT-FINDING].

Method for Perform(x) is
  [PERFORM: *x]-
    (CLASS) -> [FIELD-REPLACEABLE-UNIT]
    (OBJ) -> [TEST: C]-
              -> <PRINT> <- [OBSERVED-SYMPTOMS: *os={*}]
    (RSLT) -> [OBSERVED-SYMPTOMS: *os].
```

Figure 4.6.1 Defaults shown in the schema for "Hardware" and the method "Perform".

Conditions may be used within concepts. They may be used to represent a range of values that a concept may be equal to. For example, Figure 4.6.2 shows how a

condition could be set up within a schema graph. There are two conditions in the schema for "Person". One is the range of possible [HEIGHTs] for any "Person". The [HEIGHT] of a person must be greater than 0 inches and shorter than 96 inches. The other condition is the possible values for a person's [HAIR-COLOR]. The color of a person's hair can be any value but "Green" or "Purple". The table in Figure 4.6.3 lists conditions that can be used within concepts and a possible use for each. Conditions are in no way restricted to schema graphs. They may also be used in method and procedural attachment graphs.

```
Schema for Person(x) is
    [PERSON: *x]-
      (PTIM) -> [BIRTHDAY]-
                  (CHRC) -> [YEAR]
                  (CHRC) -> [MONTH]
                  (CHRC) -> [DAY],
      (CHRC) -> [HEIGHT: @ >0<96 inches]
      (CHRC) -> [AGE: NIL]
      (CHRC) -> [HAIR-COLOR: ~{Green|Purple}].
```

Figure 4.6.2 Defining a condition in a method.

List of Conditions

```
English                  Symbol        Use
-------                  ------       -----
equal                      =      [CITY: =Kansas City]
not equal                  ~      [TIME: ~5:30]
greater than               >      [TEMP: @ >90 F]
greater than or equal to  >=      [MONEY: @ >=$100]
less than                  <      [SPEED: @ <=55mph]
less than or equal to     <=      [FINGER: <=5]
approximation             ~=      [NUMBER: ~=50]
or                         |      [COUNTRY: ={Asia|India}]
and                        ,      [PERSON: ={Tom,Mike}]
```

Figure 4.6.3 Conditions for concepts and a possible use for each.

4.7 Daemons and Procedural Attachments

There are three types of daemons as described in section 3.6 of chapter 3. These are 1) Before and after daemons- invoked before or after a method is performed; 2) Active values- called when an instance variable's value is changed or is accessed; and 3) Procedural Attachments- activated when a value is needed, created, or removed. The before and after daemons are shown in Figure 4.7.1. Directly after the keyword "Method", the type of daemon either "before" or "after" can be specified. In Figure 4.7.1 when the diagnose method has been called, the "before" daemon will be performed before the diagnose method executes. When the diagnose method has completed the "after" daemon will be performed.

```
Method before for Diagnose(x) is
  [DIAGNOSE]-
    (OBJ) -> ["Begin diagnosing"] -> <PRINT>.

Method after for Diagnose(x) is
  [DIAGNOSE]-
    (OBJ) -> ["Diagnosing complete"] -> <PRINT>.
```

Figure 4.7.1 Before and after daemons.

Active values will be skipped since they can actually be described as procedural attachments. The procedural attachment types for the active values are described later. The example in Figure 4.7.2 describes how a procedural attachment is represented. The schema graph represents characteristics (CHRC) about a "Person". The person has a [BIRTHDAY], [HEIGHT], [HAIR-COLOR], and [AGE: NIL]. Notice there is a unique symbol "NIL" contained within the concept [AGE] in the schema and individual graphs. This denotes an attachment to the concept which means there is a procedural attachment that needs to be performed if the value is needed, removed, added, accessed, and/or replaced. The example in Figure 4.7.2 shows the procedural attachment if-needed for the concept [AGE]. If the value for the concept [AGE] is ever needed the value is calculated using the conceptual graph contained within the procedural attachment for the concept [AGE]. An example of when the value is needed is shown in Figure 4.7.2. When the individual object [PERSON: Joe] is sent to the actor <DESCRIBE-OBJECT> the value for the concept [AGE]

must be calculated. The procedural attachment calculates the age for Joe which is 40 years.

```
Schema for Person(x) is
    [PERSON: *x]-
        (PTIM) -> [BIRTHDAY]-
                     (CHRC) -> [YEAR]
                     (CHRC) -> [MONTH]
                     (CHRC) -> [DAY],
        (CHRC) -> [HEIGHT: @ inches]
        (CHRC) -> [HAIR-COLOR]
        (CHRC) -> [AGE: NIL].

Individual Person(Joe) is
    [PERSON: Joe]-
        (CHRC) -> [BIRTHDAY]-
                     (CHRC) -> [YEAR: 1946]
                     (CHRC) -> [MONTH: November]
                     (CHRC) -> [DAY: 13],
        (CHRC) -> [HEIGHT: @71 inches]
        (CHRC) -> [AGE: NIL].

procedural attachment if-needed for Age(x) is
   [AGE: *x]-
      (CLASS) -> [PERSON]
      <- <SUBTRACT> -
            <- [TIME: NOW]
            <- [YEAR: 1946] <- (CHRC) <- [BIRTHDAY].


<DESCRIBE-OBJECT> <- [PERSON: Joe].


[PERSON: Joe]-
    (CHRC) -> [BIRTHDAY]-
                 (CHRC) -> [YEAR: 1946]
                 (CHRC) -> [MONTH: November]
                 (CHRC) -> [DAY: 13],
    (CHRC) -> [HEIGHT: @71 inches]
    (CHRC) -> [HAIR-COLOR: Brown]
    (CHRC) -> [AGE: 40 years].
```

Figure 4.7.2 Schema graph for the object "Person", an individual graph of a person, and a procedural attachment that is used when the person's age is needed.

There are several options that can be used for the procedural attachments and in fact active values can be shown as procedural attachments. The active values are of two types, if-accessed and if-replaced. Other types of procedural attachments allowed are if-removed and if-added. The table below shows the allowed procedural attachment types in conceptual object-oriented programming. Each procedural attachment works on any concept within a conceptual graph.

Types of Procedural Attachments
----

1) if-needed
2) if-removed
3) if-added
4) if-accessed
5) if-replaced

4.8 Further examples

This section describes and presents a representation of the reader-writer problem using conceptual graphs and the object-oriented constructs and techniques that were described in the previous sections of Chapter 4. Figure 4.8.1 shows a type hierarchy of how the reader-writer problem could be set up. There are four objects in this problem. They are "Computer", "Card-Reader", "Line-Printer", and "Buffer". The type hierarchy in Figure 4.8.1 also shows the methods that are used on each object. Figure 4.8.2 shows the type graphs of the four objects in Figure 4.8.1.

```
                          Object
                            |
                            |
                            |
                        Computer ~~~~~ Reader
                         /  |  \        Writer
                        /   |   \       Process
                       /    |    \
  Card            Card    Line    Buffer~~~~~~ Accept-Line
  Reader~~~~~~ Reader Printer          ~~~~~~ Send-Line
  Process                 ~
                          ~
                Line-Printer-Process
```

Figure 4.8.1 Type hierarchy showing the objects defined in the reader-writer process. Also shows the methods that are performed on each object.

Type Computer(x) is [OBJECT].

Type Card-Reader(x) is [COMPUTER].

Type Line-Printer(x) is [COMPUTER].

Type Buffer(x) is [COMPUTER].

Figure 4.8.2 Type graphs using the objects in Figure 4.8.1.

The four objects outlined in the type hierarchy and the type graphs are defined with schema graphs in Figure 4.8.3. The schema for the object "Computer" contains three (PARTs) [CARD-READER's], [LINE-PRINTER's], and [BUFFER's]. There is only one method that can be invoked and it is the [READER-WRITER-PROCESS]. The schema graph for the object "Card-Reader" contains [DATA] where each piece of [DATA] has a [LINE-LENGTH]

of 80 characters. It also has a destination which is some unknown [BUFFER] and a (RATE) on how long it takes to read in [DATA]. The object "Card-Reader" has a method which is the [READER-PROCESS]. The schema for the object "Line-Printer" is similar to the object "Card-Reader" except that it has a different [LINE-LENGTH], a different method [LINE-PRINTER-PROCESS], and a different (RATE). The schema graph for the object "Buffer" contains a set of [DATA] where each piece of [DATA] has a [LINE-LENGTH] of 80 characters. The object "Buffer" has two methods, one to accept a line [ACCEPT-LINE] of [DATA] from the "Card-Reader-Process" and the other one to send a line [SEND-LINE] of [DATA] to the "Line-Printer-Process".

```
Schema for Computer(x) is
    [COMPUTER: *x]-
      (SUPER) -> [OBJECT]
      (PART) -> [CARD-READER: {*}]
      (PART) -> [LINE-PRINTER: {*}]
      (PART) -> [BUFFER: {*}]
      (MTHD) -> [READER-WRITER-PROCESS].

Schema for Card-Reader(x) is
  [CARD-READER: *x]-
     (SUPER) -> [COMPUTER]
     (OBJ) -> [DATA]-
                (CHRC) -> [LINE-LENGTH: @80 characters]
                (DEST) -> [BUFFER: *b],
     (RATE) -> [LINES: @1000 lines/min]
     (MTHD) -> [CARD-READER-PROCESS].

Schema for Line-Printer(x) is
  [LINE-PRINTER: *x]-
    (SUPER) -> [COMPUTER]
    (OBJ) -> [DATA]-
               (CHRC) -> [LINE-LENGTH: @132 characters]
               (DEST) -> [BUFFER: *b],
    (RATE) -> [LINES: @600 lines/min]
    (MTHD) -> [LINE-PRINTER-PROCESS].

Schema for Buffer(x) is
  [BUFFER: *x]-
     (SUPER) -> [COMPUTER]
     (OBJ) -> [DATA: {*}]-
                (CHRC) -> [LINE-LENGTH: @80 characters]
     (MTHD) -> [ACCEPT-LINE]
     (MTHD) -> [SEND-LINE].
```

Figure 4.8.3 Schema graphs defining the objects "computer", "card-reader", "buffer", and "line-printer".

The methods for the reader-writer problem are defined in Figure 4.8.4. To start the reader-writer process a message is sent to the object "Computer" asking it to perform the "Reader-Writer-Process". When the "Reader-Writer-Process" is performed three objects are made. These are the "Buffer", "Card-Reader", and the "Line-Printer". The actor <ASK> found in the

method "Reader-Writer-Process" asks the user to supply a unique name for each object. Next the "Reader-Writer-Process" sends a message to the [CARD-READER] asking it to perform the method [CARD-READER-PROCESS] and also at the same time a message is sent to the [LINE-PRINTER] asking it to perform the method [LINE-PRINTER-PROCESS]. The method "Card-Reader-Process" sends a message to the object "Buffer" asking it to accept a line [ACCEPT-LINE] of data from the "Card-Reader-Process" when a line of [DATA] is read in from the [CARD-READER]. After the "Buffer" has accepted the line of [DATA] the "Card-Reader-Process" tries to read in a line of [DATA] on the "Card-Reader" device. The method "Line-Printer-Process" is similar to the "Card-Reader-Process". It sends a message to the object "Buffer" asking it to perform the method [SEND-LINE]. The method [SEND-LINE] will be performed when a line of [DATA] is sent from the "Card-Reader-Process" to the "Buffer". When is gets a line of [DATA] it will be written to the [LINE-PRINTER] and again the "Line-Printer-Process" sends a message to the object "Buffer" asking it to perform the method [SEND-LINE]. The method "Accept-Line" accepts a line of [DATA] from the "Card-Reader-Process" and adds it to its set of [DATA]. The method "Send-Line" removes a line of [DATA] from the set of [DATA] and sends it back to its caller which is the "Line-Printer-Process".

```
Method for Reader-Writer-Process(x) is
  [READER-WRITER-PROCESS: *x]-
     (CLASS) -> [COMPUTER]
     <ASK> -> [BUFFER: *b] -> <MAKE-OBJECT>-
     <ASK> -> [CARD-READ: *cr] -> <MAKE-OBJECT>-
     <ASK> -> [LINE-PRINTER: *lp] -> <MAKE-OBJECT>-
     <MSGC> -
        <- [CARD-READER: *cr]-
              (MTHD) -> [CARD-READER-PROCESS]
              (OBJ) -> [DATA] -> (DEST) -> [BUFFER: *b]
        <- [LINE-PRINTER: *lp]-
               (MTHD) -> [LINER-PRINTER-PROCESS]
               (OBJ) -> [DATA] -> (DEST) -> [BUFFER: *b]
        -> ["Reader Writer Process Down"] -> <PRINT>.

Method for Card-Reader-Process(x) is
   [CARD-READER-PROCESS: *x]-
      (CLASS) -> [CARD-READER]
      <MSGC>-
        <- [BUFFER: *b]-
              (MTHD) -> [ACCEPT-LINE]
              (OBJ)  -> [DATA] <- <READ>,
        -> [CARD-READER-PROCESS]-
              (MTHD) <- [CARD-READER: *self] -> <MSGC>.

Method for Line-Printer-Process(x) is
  [LINE-PRINTER-PROCESS: *x]
    (CLASS) -> [LINE-PRINTER]
    <MSGC> -
      <- [BUFFER: *b] -> (MTHD) -> [SEND-LINE]
      -> [DATA]-
           -> <WRITE>-
                -> [LINE-PRINTER-PROCESS]-
                      (MTHD) <- [LINE-PRINTER: *self]-
                                     -> <MSGC>.

Method for Accept-Line(x) is
   [ACCEPT-LINE: *x]-
     (CLASS) -> [BUFFER]
     (OBJ) -> [DATA] -> <ADD-DATA> -
                          <- [DATA: *d={*}]
                          -> [DATA: *d={*}].

Method for Send-Line(x) is
   [SEND-LINE: *x]-
     (CLASS) -> [BUFFER]
     (OBJ) -> [DATA: *di] <- <REMOVE-DATA> -
                                <- [DATA: *d={*}]
                                -> [DATA: *d={*}].
     (RSLT) -> [DATA: *di].
```

Figure 4.8.4 The methods needed to perform the
reader-writer problem.

Chapter 4 has shown how conceptual graphs could be used to design an object-oriented system. Two examples were given, the CRIB expert system and the Reader-Writer problem. By using the conceptual graph notation a high-level knowledge representation method has been introduced providing object-oriented constructs and techniques to use them. The object-oriented constructs and techniques are not complete and the extensions and future developments will be discussed in chapter 5.

5

Conclusions

5.1 Summary and Results

Sowa's conceptual graphs have been adapted to allow the user programming capabilities in an object-oriented environment. The concepts from two different models, Sowa's conceptual graphs and object-oriented language constructs and techniques, have been integrated to form a method for conceptual object-oriented programming. This method of programming with a high-level knowledge representation method is more delineated and well defined than either Sowa's conceptual graphs or the construots and techniques of object-oriented languages. We are able to express knowledge at a very high-level without losing any meaning and while retaining the formation of a modular system. Within the method presented, concurrent processes may be expressed with little difficulty and without leaving the reader in total dismay.

5.2 Future Development

The conceptual object-oriented programming language suggested in Chapter 4 has not been implemented. Existing possibilities are to use Common Lisp as a its base language because of its open-endedness and its ability to express data as symbols along with

symbol manipulation techniques. The actors presented in chapter 4 have not been defined. These would be defined as presented in chapter 2 section 2.3.6. The relationship nodes presented in chapter 4 have not been defined with relation graphs. These would need to be defined before development could take place. There is another type of graph not used within the conceptual object-oriented programming model. This is the use of prototype graphs. Prototypes are basically used for defaults. Since we have incorporated defaults into the concepts themselves we have virtually eliminated the need for prototypes. However the use of prototypes will be left open until actual development takes place. Some of the tools that would need to be developed are listed in the table below. An artificial intelligence workstation could be used to do the full implementation of this language. They provide the standard Common Lisp programming language, a powerful graphics package to display type hierarchies, intelligent editors, capabilities of using a browser, an interpreter, a compiler, and capability to do interactive debugging. There is only one problem with the artificial intelligence workstations. They do not as yet have the capability to do concurrent programming. The machines are very fast and concurrency could be simulated. There are a number of tools that will be required to make this implementation useful including those listed below.

Tools

-----

1. Graphics package to show the type hierarchy.
2. Editor for program creation and modification.
3. Browser for viewing the program's objects and their relationships.
4. Interpreter for test execution.
5. Compiler for production programs (and for efficiency).
6. Interactive debugger for understanding faulty programs.

5.3 Comparison to other Research

Although there is little difference between the basic notation of this model and other knowledge representation methods, the notation for this model has been modified and configured in a way to allow for object-oriented programming. The work here is comparable to the work of Roger Hartley, Hartley[1985], in which he uses Sowa's conceptual graphs to show procedural knowledge for expert systems. His method made use of several actors and an extensive set of relation nodes. The method in this research used a few actors and a minimal amount of new relation nodes to define the conceptual object-oriented programming environment. This method is simpler, more understandable, and clearer than Hartley's. In Figure 5.3.1 a schema graph definition from Hartley's work is presented. The schema graph describes how the "diagnose" for computer fault finding is performed in the expert system CRIB. Hartley's "diagnose" schema graph can be compared to

the "diagnose" method graph defined by conceptual object-oriented constructs and techniques in chapter 4. The method graph is presented again, in Figure 5.3.2, for comparison of Hartley's work and the work presented in this paper.

```
Schema for Diagnose(x) is
 [EVENT: [DIAGNOSE: *x] -
    (OBJ) -> [FUNCTION: system]-
                (LOC) -> [UNIT: computer]-
                           (ATTR) -> [FAULTY],
                (ATTR) -> [CURRENT]]-
  (TRIG) -> <N>-
            (IPC) <- [FUNCTION: system]
            (THEN) ->
 [EVENT: *ch=[CHOOSE]-
  (OBJ)->[TEST: *t]-
         (INST)<-[DISCRIMINATE]-
                 (MANR)->[BEST]
                 (OBJ)->[FUNCTION]-
                         (SUBSUMES)<-[FUNCTION: *f]-
                         (ATTR)->[CURRENT]]-
  (TRIG) -> <N>-
            (IPC) <- [TEST: *t]
            (THEN) ->
 [EVENT: [PERFORM]-
            (OBJ) -> [TEST: *t]
            (RLST) -> [OBSERVED-SYMPTOM: *os={*}]]-
  (TRIG) -> <N>-
            (IPC) <- [OBSERVED-SYMPTOM: *os]
            (THEN) ->
 [EVENT: [ADD]-
           (OBJ) -> [OBSERVED-SYMPTOM: *os]
           (DEST) -> [CURRENT-SYMPTOM: {*}]
           (RSLT) -> [CURRENT-SYMTPOM: *cs={*}]]-
  (TRIG) -> <N>-
            (IPC) <- [CURRENT-SYMPTOM: *cs]
            (THEN) ->
  [EVENT: [ANALYSE]-
            (INST) -> [CURRENT-SYMPTOM: *cs]
            (OBJ) -> [FUNCTION: *f]]-
  (TRIG) -> <N> -> [EVENT: *ch].
```

Figure 5.3.1 Work done By Roger Hartley. Shows inferencing for the diagnose operation.

```
Method for Diagnose(x) is
  [DIAGNOSE: *x]-
    (CLASS) -> [HARDWARE]
    (MTHDCLASS) -> [COMPUTER-FAULT-FINDING]
    <MSGC>-
      <- [FIELD-REPLACEABLE-UNIT]-
           (MTHD) -> [CHOOSE],
      -> [TEST: *ts]-
           (OBJ) <- [FIELD-REPLACEABLE-UNIT: *fru]-
    <MSGC>-
      <- [FIELD-REPLACEABLE-UNIT: *fru]-
           (MTHD) -> [PERFORM]
           (OBJ)  -> [TEST: *ts],
      -> [OBSERVABLE-SYMPTOMS: *os={*}] -
           -> <ADD-SETS>-
                <- [CURRENT-SYMPTOMS: *cs={*}]
                -> [CURRENT-SYMPTOMS: *cs]-
    <MSGC>-
      <- [FIELD-REPLACEABLE-UNIT: *fru]-
             (MTHD) -> [ANALYSE]
      -> [FIELD-REPLACEABLE-UNIT: *fru]-
             (ATTR) -> [FAULTY] -> <HALT>
      -> [DIAGNOSE] -> (MTHD) <- [HARDWARE: *self]-
                                               -> <MSGC>
    (RSLT) -> [FIELD-REPLACEABLE-UNIT: *fru]-
                     (ATTR) -> [FAULTY].
```

Figure 5.3.2 Conceptual Object-Oriented method for "diagnose".

The language defined in this research can be compared to the existing languages of object-oriented programming. Smalltalk, Flavors, Loops, Common Loops and others lack the expressibility of representing knowledge at a very high-level. Also these languages are limited in that they are restrictive in nature when the user needs to express knowledge about concurrent processes. In Figure 5.3.3 an example using Flavors is presented. The example shows how some of the objects from CRIB could be represented using Flavors. The objects defined with the "defflavor" function are "com-

puter", "hardware", and "field-replaceable-unit". The method for the "diagnose" operation is also defined with the Flavors "defmethod" function. The object-oriented programming lenguages Smalltalk, Loops, and CommonLoops represent knowledge similer to thet of Flavors, therefore there is no need to show examples using each language's syntax.

```
(defflavor computer (serial-number)
                    (physical-object))

(defflevor herdware (current-symptoms
                     test
                     time
                     fru-test '(xerox-1186-1-1a)
                    (computer))

(defflavor field-replaceable-unit (invalid-symptoms
                                   current-symptoms
                                   test
                                   time))

(defmethod (hardware :diagnose) ()
  (setq fru-test (send (cer fru-test) ':choose))
  (setq current-symptoms
      (add-sets (send (car fru-test) ':perform)
                current-symptoms))
  (cond ((null (send (cer fru-test) ':analyse
                              (cadr fru-test)))
          (send (cer fru-test) ':diagnose))
        (t (msg "Faulty Field Repleceable Unit is "
                (car fru-test)))))
```

Figure 5.3.3 Flavors examples describing objects and methods thet ere in CRIB.

Sowa's definition of conceptual grephs is very strong at the abstrect level but lacks detail in many arees. Clancey, Clancey[1985] in his review, of Sowa's work says "Sowa has provided e clean well-grounded notation for knowledge representetion that many

researchers will want to emulate and build upon." He then goes on to present his views on the areas where Sowa's book is lacking, "...but his knowledge of both expert systems and cognitive science issues is not complete..." and "...the relation of conceptual graphs to heuristic reasoning is not adequately developed or demonstrated by working programs." This represents the state of Sowa's theory rather than the book itself. A follow up to Sowa's book providing examples fulfilling the above deficiencies is needed by those researchers who are trying to emulate and build upon his notation. We are unable to present any examples from Sowa's book that would be comparable to the work defined within this paper. The reason for this is because Sowa does not clearly define the constructs and techniques to do inferencing or heuristic reasoning using conceptual graphs.

REFERENCES

[1] Allen, E., Trigg , R., & Wood, R., Franz Lisp Environment. The Maryland Artificial Intelligence Group, University of Maryland, 1983.

[2] Bobrow, D.G., Kahn, K.M., & Stefik, M., 'Integrating Access-Oriented Programming into a Multiparadigm Environment.' To Appear in IEEE Software.

[3] Bobrow, D.G. & Stefik, M., 'Object-Oriented Programming: Themes and Variations.' AI Magazine, Volume 6 Number 4, Winter 1986.

[4] Bobrow, D., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., and Zdybel, F., 'CommonLoops: Merging Common Lisp and Object-Oriented Programming', Xerox PARC, Pre IJCAI-85 Draft, ISL-85-8, 1985.

[5] Bobrow, D., & Stefik, Mark, The LOOPS Manual. Xerox PARC, December 1983.

[6] Brachman, R.J., An Essential Hybrid Reasoning System: Knowledge and Symbol Level Accounts of KRYPTON. IJCAI-85 at UCLA, August 1985.

[7] Cattell, R., 'The Structure of Intelligence in Relation to the Nature-Nuture Controversy.' in Intelligence genetic and environmental influences. R. Cancro (Eds). New York : Grune and Stratton, 1971.

[8] Clancey, W.J., Book Review, Conceptual Structures: Information Processing in Mind and Machine, by John Sowa. Artificial Intelligence Journal, Number 27, September 1985, 113-128.

[9] Federaro, J., Sklower, K., & Layer, K., The Franz Lisp manual, June 1983.

[10] Fikes, R. & Kehler, T., 'The Role of Frame-Based Representation in Reasoning.' Communications of the ACM, Volume 28 Number 9, September 1985.

[11] Genesereth, M., An Overview of Meta-Level Architecture. Stanford University, Computer Science Department. Stanford, California, 1984.

[12] Genesereth, M., Partial Programs. Stanford University, Computer Science Department. Stanford, California, 1984.

[13] Hansen, Per Brinch, The Architecture of Concurrent Programs. Prenctice-Hall, New Jersey, 1977.

[14] Hartley, Roger T., 'Representation of Procedural Knowledge for Expert Systems.' Department of Computer Science, Kansas State University, 1985.

[15] Hewitt, C. & Baker, H. Jr. 'Actors and Continuous Functionals.' Massachusetts Institute of Technology, Laboratory for Computer Science, MIT/LCS/TR-194, Cambridge, MA., 1977.

[16] Horowitz, Ellis, 'Object-Oriented Programming Languages' in Fundamentals of Programming Languages, Second Edition. Computer Science Press, 1984, 395-418.

[17] Minsky, Marvin., 'A Framework for Representing Knowledge.' in The Psychology of Computer Vision. McGraw-Hill, 1975, 211-280.

[18] Rich, Elaine, Artificial Intelligence, McGraw-Hill New York, 1983.

[19] Schank, Roger C. & Childers, Peter, The Cognitive Computer: On Language, Learning, and Artificial Intelligence, 1984.

[20] SmallTalk-80 Programming Language, BYTE, Volume 6 Number 8, August 1981.

[21] Snyder, Alan, Object-Oriented Programming for CommonLisp. ATC-85-1. Palo Alto, CA: Hewlett Packard Laboratories, 1985.

[22] Steele, Guy Jr., Common Lisp: The Language, Digital Press, 1984.

[23] Sowa, John F., Conceptual Structures: Information Processing in Mind and Machine, Addison Wesley, Reading, MA., 1984.

[24] Sowa, John F., 'A Conceptual Schema for Knowledge-Based Systems.' ACM, 1980, 193-195.

[25] Unger, Elizabeth, A., 'A Natural Model for Concurrent Computation', University of Kansas, 1978.

[26] Weinreb, D. and Moon, D., 'FLAV Objects, Message Passing, and Flavors.' in Lisp Machine Manual. Symbolics Inc., Cambridge, MA., 1984.

[27] Winograd, Terry, 'Frame Representations and the Declarative/Procedural controversy.' in Representation and Understanding : Studies in Cognitive Science. Daniel G. Bobrow and Allan Collins (Eds). New York : Academic Press, 1975, 185-210.

[28] Winston, Patrick Henry, Artificial Intelligence, second edition, Addison Wesley, Reading, MA., 1984.

[29] Winston, Patrick & Horn, Berthold-Klaus, LISP, First edition (1981) and Second edition (1984), Addison-Wesley.

CONCEPTUAL OBJECT-ORIENTED PROGRAMMING

by

TIMOTHY R. HINES

B.S., Kansas State University, 1983

AN ABSTRACT OF A MASTER'S THESIS

Submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

ABSTRACT

Object-oriented programming is growing in popularity because of its modular programming constructs and of the techniques available for use with them. The object-oriented languages available at the present time use programming techniques that are at the logical representation level in the knowledge levels. Languages such as Flavors, Loops, Smalltalk, and CommonLoops are used at the logical representation level when implementing an object-oriented system, that is one are constrained to using a low level representation form when programming. To represent knowledge at a higher level, one needs to represent the knowledge in an object-oriented programming language using either a conceptual or natural language representation method.

The technique of designing conceptual structures is represented at the conceptual level in the knowledge levels. John Sowa's Conceptual Structures is the method chosen to represent the formal constructs of the objects and the techniques to use them. This paper presents the formal design of the objects and their attributes using conceptual structures. The integration within this model of object-oriented techniques with conceptual representation methods provides the advantages of each technique. The environment of the system is described, with emphasis on message passing and reasoning between objects. This system allows the programmer more freedom designing and implementing an object-oriented system than currently available systems.